

Скотт Миллетт



Ник Тьюн

ПРЕДМЕТНО- ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

ПАТТЕРНЫ, ПРИНЦИПЫ И МЕТОДЫ





Patterns, Principles, and Practices of Domain-Driven Design

Scott Millett

Nick Tune



СКОТТ МИЛЛЕТТ, НИК ТЬЮН

ПРЕДМЕТНО- ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

ПАТТЕРНЫ, ПРИНЦИПЫ И МЕТОДЫ



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2017

Предметно-ориентированное проектирование: паттерны, принципы и методы

Серия «Для профессионалов»

Перевел с английского А. Киселев

Заведующая редакцией
Ведущий редактор
Литературный редактор
Художественный редактор
Корректоры
Верстка

Ю. Сергиенко
Н. Римицан
Е. Самородских
С. Заматевская
С. Беляева, Н. Викторова
Л. Егорова

ББК 32.973.2-018

УДК 004.42

Миллетт С., Тьюн Н.

М60 Предметно-ориентированное проектирование: паттерны, принципы и методы. —
СПб.: Питер, 2017. — 832 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-01984-2

Писать программы легко — во всяком случае, с нуля. Но изменить однажды написанный программный код, который создали другие разработчики или вы сами каких-то шесть лет тому назад, — гораздо сложнее. Программа работает, но вы не знаете точно, как именно. Даже обращение к экспертам в предметной области ничего не даст, поскольку в коде не сохранилось никаких следов привычного для них языка.

Предметно-ориентированное проектирование (Domain-Driven Design, DDD) — это процесс тесной увязки программного кода с реалиями предметной области.

Благодаря ему добавление в программный продукт новых возможностей по мере его развития становится таким же простым, как и при создании программы с нуля.

Эта книга в полной мере соответствует философии DDD и позволяет разработчикам перейти от философских рассуждений к решению практических задач.

Она делится на четыре части: часть I посвящена философии, принципам и приемам предметно-ориентированного проектирования; в части II подробно обсуждаются стратегические шаблоны интеграции ограниченных контекстов; часть III охватывает тактические шаблоны создания эффективных моделей предметной области; часть IV в деталях описывает шаблоны проектирования, которые позволяют извлекать пользу из модели предметной области и создавать эффективные приложения.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1118714706 англ.

© 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

ISBN 978-5-496-01984-2

© Перевод на русский язык ООО Издательство «Питер», 2017

© Издание на русском языке, оформление ООО Издательство «Питер», 2017

© Серия «Для профессионалов», 2017

Права на издание получены по соглашению с John Wiley & Sons Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Изготовлено в России. Изготовитель: ООО «Питер Пресс». Место нахождения и фактический адрес:

192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 06.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 08.06.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 67,080. Тираж 1000. Заказ 4353

Отпечатано в АО «Первая Образцовая типография» филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

Краткое содержание

Часть I. Принципы и приемы предметно-ориентированного проектирования	33
Глава 1. Что такое предметно-ориентированное проектирование?.....	34
Глава 2. Дистилляция предметной области задачи.....	48
Глава 3. Концентрация на смысловом ядре	68
Глава 4. Проектирование на основе модели	81
Глава 5. Шаблоны реализации предметной модели.....	102
Глава 6. Обеспечение целостности моделей предметной области с помощью ограниченных контекстов	118
Глава 7. Карты контекстов	137
Глава 8. Архитектура приложения.....	152
Глава 9. Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование	170
Глава 10. Применение принципов, приемов и шаблонов DDD	183
Часть II. Стратегические шаблоны: взаимодействие ограниченных контекстов.....	205
Глава 11. Введение в интеграцию ограниченных контекстов	206
Глава 12. Интеграция посредством обмена сообщениями	241
Глава 13. Интеграция с RPC и REST посредством HTTP	313
Часть III. Тактические шаблоны: создание эффективных моделей предметной области.....	379
Глава 14. Знакомство со стандартными блоками моделирования предметной области	380

Глава 15. Объекты-значения.....	402
Глава 16. Сущности.....	436
Глава 17. Службы предметной области	468
Глава 18. События предметной области	485
Глава 19. Агрегаты.....	508
Глава 20. Фабрики	555
Глава 21. Репозитории	565
Глава 22. Регистрация событий.....	684
 Часть IV. Шаблоны проектирования эффективных приложений ...	 735
Глава 23. Конструирование пользовательских интерфейсов приложения... 	736
Глава 24. CQRS: архитектура ограниченного контекста	760
Глава 25. Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования.....	780
Глава 26. Запросы: предметная отчетность.....	807

Оглавление

Об авторе.....	20
О соавторе.....	20
О научном редакторе	20
Благодарности	21
Введение.....	22
Обзор книги и технологии	22
Пространство задачи.....	23
Пространство решений	23
Структура этой книги.....	24
Часть I. Принципы и приемы предметно-ориентированного проектирования.....	25
Часть II. Стратегические шаблоны: взаимодействие ограниченных контекстов.....	27
Часть III. Тактические шаблоны: создание эффективных моделей предметной области	27
Часть IV. Шаблоны проектирования эффективных приложений.....	29
Кому адресована эта книга.....	30
Исходный код.....	30
Опечатки	30
Форумы P2P.WROX.COM	31
Резюме.....	32
От издательства	32

Часть I. Принципы и приемы предметно-ориентированного проектирования 33

Глава 1. Что такое предметно-ориентированное проектирование?.....	34
Проблемы разработки программного обеспечения для сложных предметных областей.....	35
Код, созданный без использования общего языка	36
Недостаточная организация.....	36
Шаблон «Большой ком грязи» тормозит развитие продукта	37
Недостаточное внимание к предметной области	37
Как шаблоны предметно-ориентированного проектирования помогают обуздать сложность	38
Стратегические шаблоны DDD	38
Тактические шаблоны DDD.....	41
Пространство задачи и пространство решения	42
Приемы и принципы предметно-ориентированного проектирования	42
Концентрация на смысловом ядре.....	43
Изучение через сотрудничество.....	43
Создание моделей посредством исследований и экспериментов	44
Коммуникация	44
Понимание границ применимости модели	44
Постоянное развитие модели.....	45

Типичные заблуждения, связанные с предметно-ориентированным проектированием.....	45
Суть DDD — в тактических шаблонах	45
DDD — это фреймворк.....	46
DDD — это панацея	46
Ключевые идеи.....	47
Глава 2. Дистилляция предметной области задачи.....	48
Переработка знаний и сотрудничество	48
Достижение единого понимания через общий язык.....	49
Важность знаний в предметной области.....	50
Роль бизнес-аналитиков.....	51
Непрекращающийся процесс.....	51
Углубление знаний с помощью экспертов в предметной области.....	52
Эксперты в предметной области и заинтересованные лица	52
Углубление знаний специалистов со стороны бизнеса.....	52
Тесное взаимодействие с экспертами в предметной области	52
Шаблоны эффективной переработки знаний	54
Сосредоточьтесь на самых интересных темах.....	54
Начните с вариантов использования	54
Задавайте продуктивные вопросы	54
Схемы и диаграммы	55
Используйте CRC-карточки	56
Не спешите именовать понятия в модели	56
Разработка через реализацию поведения	57
Быстрое прототипирование.....	58
Изучайте «бумажные» процессы.....	59
Ищите уже созданные модели	59
Разберитесь в истинных намерениях.....	60
Событийный штурм	60
Составление карты воздействий.....	61
Бизнес-модель организации.....	63
Целенаправленное открытие.....	64
Водоворот исследования модели.....	65
Ключевые идеи.....	66
Глава 3. Концентрация на смысловом ядре	68
Зачем нужна декомпозиция предметной области.....	68
Как выделить суть задачи	69
Старайтесь понять, что стоит за требованиями.....	69
Составьте обзор предметной области, в котором обозначены ключевые моменты	70
Как сосредоточиться на главной задаче	71
Дистилляция предметной области задачи	72
Смысловое ядро	74
Рассматривайте смысловое ядро как продукт, а не как проект	75
Неспециализированные области	75
Поддерживающие области.....	76
Как деление на подобласти формирует контур решения	76
Не все части системы требуют детальной проработки.....	76
Уделяйте больше внимания ясности границ, а не совершенству моделей	77
Реализация смыслового ядра не обязана быть идеальной с самого начала.....	78
Создавайте подобласти с прицелом на замену, а не на повторное использование.....	79
А что, если смысловое ядро отсутствует?.....	79
Ключевые идеи.....	79

Глава 4. Проектирование на основе модели	81
Что такое предметная модель?	82
Предметная область и ее модель	82
Аналитическая модель	83
Программная модель	83
Программная модель служит основным представлением предметной модели	84
Проектирование на основе модели	84
Проблемы, возникающие при заблаговременном проектировании	85
Командное моделирование	86
Использование единого языка для связывания аналитической и программной моделей	88
Язык переживет вашу программу	89
Язык бизнеса	89
Перевод между языком разработчиков и бизнес-языком	89
Совместная работа над созданием единого языка	90
Шлифовка языка на конкретных примерах	91
Учите экспертов в предметной области сосредоточиваться на задаче и не переходить к ее решению	92
Эффективные приемы формирования языка	93
Как создавать эффективные предметные модели	94
Жертвуйте точностью, если она стоит на пути к хорошей модели	95
Включайте в модель только то, что имеет отношение к задаче	96
Предметные модели полезны лишь временно	97
Используйте недвусмысленную терминологию	97
Ограничивайте свои абстракции	97
Применяйте абстракции на правильно выбранном уровне	98
Описывайте в абстракциях поведение, а не реализацию	98
Воплощайте модель в коде как можно раньше и чаще	99
Не останавливайтесь на первой же работоспособной идее	99
Когда следует применять проектирование на основе модели	99
Не тратьте силы на моделирование, если это того не стоит	100
Сосредоточьтесь на смысловом ядре	100
Ключевые идеи	101
Глава 5. Шаблоны реализации предметной модели	102
Загружаемые примеры кода для этой главы	102
Уровень предметной области	103
Шаблоны реализации предметной модели	104
Предметная модель	105
Сценарий транзакции	109
Модуль таблицы	111
Активная запись	112
Анемичная предметная модель	112
Анемичная предметная модель и функциональное программирование	113
Ключевые идеи	117
Глава 6. Обеспечение целостности моделей предметной области с помощью ограниченных контекстов	118
Проблемы архитектур с единственной моделью	119
Сложность модели может увеличиваться	119
Работа нескольких групп над единственной моделью	120
Неоднозначность языка модели	121
Применимость предметных понятий	121

Интеграция с унаследованным или сторонним кодом	124
Предметная модель не является моделью предприятия	124
Использование ограниченных контекстов для декомпозиции больших моделей	125
Определение границ модели	127
Создание контекстов на основе организации разработчиков	129
Различия между подобластями и ограниченными контекстами	132
Реализация ограниченных контекстов	132
Ключевые идеи	136
Глава 7. Карты контекстов	137
Карта реальности	138
Техническая действительность	139
Организационная действительность	139
Отображение актуальной действительности	141
Выделение смыслового ядра на карте	141
Определение отношений между ограниченными контекстами	141
Предохранительный слой	141
Общее ядро	142
Служба с открытым протоколом	143
Отдельное существование	144
Партнерство	145
Отношения «вышестоящий/нижестоящий»	145
Отношения на карте контекстов	147
Стратегическая важность карт контекстов	147
Сохранение целостности	148
Основа для планирования работ	148
Понимание принадлежности и ответственности	149
Вскрытие запутанных областей в рабочих процессах на предприятии	149
Выявление нетехнических преград	149
Способствование налаживанию общения	150
Ускорение адаптации новых разработчиков	150
Ключевые идеи	150
Глава 8. Архитектура приложения	152
Загружаемые примеры кода для этой главы	152
Архитектура приложения	152
Разделение задач, решаемых приложением	153
Отделение от сложностей предметной области	153
Многоуровневая архитектура	153
Инверсия зависимостей	154
Предметный уровень	155
Уровень прикладных служб	155
Инфраструктурные уровни	156
Взаимодействия между уровнями	156
Тестирование в изоляции	157
Не используйте схему данных, общую для всех ограниченных контекстов	157
Сравнение архитектур приложений и ограниченных контекстов	158
Прикладные службы	160
Прикладная и предметная логика	162
Определение и экспортирование функций	162
Координация выполнения сценариев использования	163
Прикладные службы представляют сценарии использования, а не CRUD-операции	164

Предметный уровень как детали реализации	164
Отчеты о состоянии предметной модели.....	164
Модели чтения и транзакционные модели	165
Клиенты приложения.....	166
Ключевые идеи.....	168
Глава 9. Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование	170
Переоценка важности тактических шаблонов.....	171
Использование одной архитектуры для всех ограниченных контекстов.....	171
Идеализация тактических шаблонов	171
Ошибочное принятие строительных блоков за ценность DDD	172
Сосредоточенность на коде, а не на принципах DDD.....	172
Недооценка истинной ценности DDD: сотрудничество, общение и контекст.....	173
Получение «большого кома грязи» из-за недопонимания важности контекста	174
Появление неоднозначности и недопонимания из-за неудач при создании единого языка	174
Получение сугубо технических решений из-за недостатка взаимодействия	175
Большие затраты времени на то, что не представляется важным	176
Превращение простых задач в сложные	176
Применение принципов DDD к простым областям, не имеющим большого значения для предприятия.....	177
Отказ от шаблона CRUD как от антишаблона	177
Использование шаблона «Предметная модель» для всех ограниченных контекстов.....	178
Спросите себя: нужна ли эта дополнительная сложность?.....	178
Недооценка стоимости применения DDD	178
Попытка добиться успеха без мотивированных и целеустремленных разработчиков.....	178
Сотрудничество с незаинтересованными специалистами.....	179
Методологии итерационной разработки.....	179
Применение DDD к любым задачам.....	180
Жертвование простотой ради ненужной чистоты.....	180
Пустая трата сил на поиски подтверждений правоты	180
Постоянное стремление к совершенству кода	181
Цель DDD — обеспечить ценность.....	181
Ключевые идеи.....	182
Глава 10. Применение принципов, приемов и шаблонов DDD	183
Загружаемые примеры кода для этой главы	183
Внедрение DDD	183
Обучение разработчиков	184
Общение со специалистами.....	184
Применение принципов DDD	185
Понимание замысла	185
Определение требуемой функциональности	186
Понимание действительной картины	187
Моделирование решения.....	188
Исследования и эксперименты	196
Ставьте свои предположения под сомнение.....	197
Моделирование — это текущая работа	197
Не существует неправильных моделей	197
Податливый код способствует раскрытию	198

Превращение неявного в явное	198
Борьба с неоднозначностями	199
Давайте названия	201
Сначала решение задачи, и только потом ее реализация	201
Не решайте все задачи	201
Как узнать, что все делается правильно?	202
Не стремитесь к идеалу	202
Практика, практика и еще раз практика	203
Ключевые идеи	203

Часть II. Стратегические шаблоны: взаимодействие ограниченных контекстов..... 205

Глава 11. Введение в интеграцию ограниченных контекстов 206

Загружаемые примеры кода для этой главы	206
Как интегрировать ограниченные контексты	207
Ограниченные контексты независимы	208
Проблемы интеграции ограниченных контекстов на уровне программного кода	208
Использование физических границ для гарантий чистоты моделей	213
Интеграция с унаследованными системами	214
Экспортирование унаследованных систем в виде служб	216
Интеграция распределенных ограниченных контекстов	218
Стратегии интеграции распределенных ограниченных контекстов	218
Интеграция через базу данных	219
Интеграция через простые файлы	220
RPC	221
Обмен сообщениями	222
REST	223
Проблемы применения DDD в распределенных системах	223
Проблемы с RPC	224
Распределенные транзакции ухудшают масштабируемость и надежность	227
Реактивная философия DDD и управление по событиям	229
Проблемы и компромиссы асинхронных сообщений	231
Технология RPC все еще актуальна?	232
Реактивная философия DDD и SOA	233
Представление ограниченных контекстов в виде служб SOA	234
Еще шаг вперед с архитектурой микрослужб	238
Ключевые идеи	240

Глава 12. Интеграция посредством обмена сообщениями 241

Загружаемые примеры кода для этой главы	241
Основы обмена сообщениями	242
Шина сообщений	243
Надежный обмен сообщениями	244
Сохранить и передать	245
Команды и события	245
Потенциальная непротиворечивость	246
Создание приложения электронной коммерции с применением NServiceBus	247
Проектирование системы	248
Отправка команд из веб-приложения	253
Отправка команд	258
Обработка команд и публикация событий	262

Увеличение надежности внешних вызовов HTTP с помощью шлюзов сообщений	270
Потенциальная непротиворечивость на практике	278
Ограниченные контексты хранят все необходимые им данные локально	280
Объединяем все вместе в пользовательском интерфейсе	289
Сопровождение приложений, использующих обмен сообщениями	292
Поддержка версий сообщений	292
Мониторинг и масштабирование	298
Интеграция ограниченных контекстов с применением MASS TRANSIT	301
Мост обмена сообщениями	303
Mass Transit	303
Ключевые идеи	311
Глава 13. Интеграция с RPC и REST посредством HTTP	313
Загружаемые примеры кода для этой главы	313
Преимущества HTTP	315
Независимость от выбора платформы	315
HTTP понятен любому	315
Множество проверенных инструментов и библиотек	316
Возможность пользоваться своими же API	316
RPC	317
Реализация RPC через HTTP	317
Выбор варианта RPC	333
REST	334
Разоблачение мифов REST	334
REST для интеграции ограниченных контекстов	339
Сопровождение приложений REST	375
Недостатки интеграции ограниченных контекстов с применением REST	376
Ключевые идеи	378
 Часть III. Тактические шаблоны: создание эффективных моделей предметной области	 379
Глава 14. Знакомство со стандартными блоками моделирования предметной области	380
Загружаемые примеры кода для этой главы	380
Тактические шаблоны	381
Шаблоны моделирования предметной области	382
Сущности	382
Объекты-значения	385
Предметные службы	388
Модули	389
Шаблоны жизненного цикла	390
Агрегаты	391
Фабрики	394
Репозитории	395
Другие шаблоны	396
Предметные события	396
Регистрация событий	398
Ключевые идеи	399

Глава 15. Объекты-значения	402
Загружаемые примеры кода для этой главы	402
Когда использовать объекты-значения	403
Представление описательного понятия, не имеющего индивидуальности	403
Улучшение ясности	404
Характерные особенности	406
Отсутствие индивидуальности	406
Сравнение по атрибутам	407
Разнообразие возможностей	411
Согласованность	411
Неизменяемость	411
Комбинируемость	413
Автоматическая проверка	415
Простота тестирования	418
Общие шаблоны моделирования	420
Статические фабричные методы	420
Микротипы	421
Отказ от коллекций	424
Сохранение	427
NoSQL	427
SQL	428
Ключевые идеи	435
Глава 16. Сущности	436
Загружаемые примеры кода для этой главы	436
Понимание сущностей	437
Предметные понятия с индивидуальностью и жизненным циклом	437
Зависимость от контекста	438
Реализация сущностей	438
Идентификация	438
Включение логики в объекты-значения и предметные службы	445
Проверка и соблюдение правил	448
Сосредоточьтесь на поведении, а не на данных	451
Избегайте ошибки «моделирования реального мира»	454
Проектирование для распределенных окружений	455
Основные принципы и шаблоны моделирования сущностей	457
Реализуйте проверки и инварианты с помощью спецификаций	457
Избегайте шаблона «Состояние»; используйте явное моделирование	460
Избегайте использования методов чтения/записи с шаблоном «Хранитель»	464
Отдавайте предпочтение функциям без побочных эффектов	465
Ключевые идеи	466
Глава 17. Службы предметной области	468
Загружаемые примеры кода для этой главы	468
Предметные службы	469
Когда использовать предметные службы	469
Внутреннее устройство предметной службы	474
Избегайте шаблона «Анемичная предметная модель»	475
Отличие от прикладных служб	475
Применение предметных служб	476
В уровне служб	476
В предметной области	477
Ключевые идеи	484

Глава 18. События предметной области	485
Загружаемые примеры кода для этой главы	485
Суть шаблона «Предметные события»	486
Важные предметные происшествия, которые уже случились	486
Реакция на события	487
Возможная асинхронность	487
Внутренние и внешние события	488
Обработка событий	490
Вызов предметной логики	490
Вызов прикладной логики	490
Варианты реализации шаблона «Предметные события»	490
Используйте модель событий .Net Framework	491
Использование шины памяти	493
Статический класс DomainEvents Уди Дахана	496
Возврат предметных событий	499
Использование контейнера IoC в качестве диспетчера событий	502
Тестирование предметных событий	503
Модульное тестирование	503
Тестирование прикладной службы	505
Ключевые идеи	506
Глава 19. Агрегаты	508
Загружаемые примеры кода для этой главы	508
Управление сложными деревьями объектов	509
Однонаправленность предпочтительнее	509
Ограничение связей	512
Идентификаторы объектов предпочтительнее ссылок	513
Агрегаты	515
Проектирование на основе предметных инвариантов	517
Более высокий уровень абстракции предметной области	517
Маленькие агрегаты предпочтительнее	523
Определение границ агрегатов	525
eBidder: интернет-аукцион	525
Согласование с инвариантами	527
Согласование с транзакциями	530
Игнорируйте требования пользовательского интерфейса	531
Избегайте простых коллекций и контейнеров	532
Не зацикливайтесь на отношениях владения	532
Реорганизация агрегатов	533
Главное — потребности сценариев использования, а не соответствие реальности	533
Реализация агрегатов	534
Выбор корня агрегата	534
Ссылка на другие агрегаты	538
Реализация хранения	542
Реализация транзакционной согласованности	547
Реализация потенциальной согласованности	548
Поддержка одновременного доступа	551
Ключевые идеи	553
Глава 20. Фабрики	555
Загружаемые примеры кода для этой главы	555
Роль фабрики	556
Отделение использования от создания	556

Скрытие внутренних деталей	556
Скрытие решения о выборе типа создаваемого объекта	559
Фабричные методы в агрегатах	560
Фабрики для восстановления	562
Практическое использование фабрик	563
Ключевые идеи	564
Глава 21. Репозитории	565
Загружаемые примеры кода для этой главы	565
Репозитории	565
Ошибочные представления о шаблоне	567
«Репозиторий» — это антишаблон?	568
Отличия между предметной моделью и моделью хранения	568
Обобщенный репозиторий	569
Стратегии хранения агрегатов	573
Использование фреймворка сохранения, способного отобразить предметную модель в модель данных без компромиссов	573
Использование фреймворка сохранения, не способного отобразить предметную модель без компромиссов	574
Общедоступные методы чтения и записи	574
Шаблон «Хранитель»	576
Потоки событий	578
Прагматизм	578
Репозиторий — четко определенный контракт	579
Управление транзакциями и единицы работы	580
Сохранять или не сохранять	584
Фреймворки сохранения, автоматически определяющие изменения в предметных объектах	585
Необходимость явно сохранять агрегаты	586
Репозиторий как предохранительный слой	587
Другие области ответственности репозитория	588
Генерирование идентификаторов сущностей	588
Обобщенные сведения о коллекциях	590
Одновременный доступ	591
Контрольные проверки	594
Антишаблоны реализации репозитория	595
Антишаблон: поддержка универсальных запросов	595
Антишаблон: отложенная загрузка	596
Антишаблон: использование репозитория для составления отчетов	596
Реализации репозитория	597
Фреймворк сохранения способен отобразить предметную модель в модель данных без компромиссов	598
Фреймворк сохранения не способен отобразить предметную модель в модель данных без компромиссов	646
Ключевые идеи	683
Глава 22. Регистрация событий	684
Загружаемые примеры кода для этой главы	684
Ограничения на хранение состояния в виде моментального снимка	685
Конкурентные преимущества от хранения состояния в виде потока событий	686
Временные запросы	687
Проекция	688
Моментальные снимки	689

Агрегаты с поддержкой регистрации событий	689
Структурирование	690
Сохранение и восстановление	694
Решение проблем одновременного доступа	698
Тестирование	700
Создание хранилища событий	701
Проектирование структуры хранилища	702
Создание потоков событий	703
Добавление событий в потоки	704
Извлечение потоков событий	704
Добавление поддержки моментальных снимков	705
Управление одновременным доступом	707
Хранилище событий на основе SQL Server	711
Оправданно ли создание собственного хранилища событий?	717
Использование специализированного хранилища событий	718
Установка Event Store	718
Использование клиентской библиотеки для C#	718
Временные запросы	723
Создание проекций	726
Шаблон CQRS и регистрация событий	728
Использование проекций для создания кэшированных представлений	729
Объединение CQRS с приемом регистрации событий	730
Еще раз о выгодах регистрации событий	730
Конкурентные преимущества для предприятия	731
Выразительность поведения агрегатов	731
Простота хранения	731
Простота отладки	732
Оценка ценности регистрации событий	732
Поддержка версий	732
Необходимость освоения новых понятий и обретения навыков	732
Необходимость изучения новых технологий и овладения ими	733
Более строгие требования к емкости хранилища данных	733
Дополнительные ресурсы	733
Ключевые идеи	733

Часть IV. Шаблоны проектирования эффективных приложений ... 735

Глава 23. Конструирование пользовательских интерфейсов приложения	736
Загружаемые примеры кода для этой главы	736
Основы проектирования	737
Единые и составные пользовательские интерфейсы	737
HTML API и Data API	740
Координация/компоновка на стороне клиента и сервера	741
Пример 1: Пользовательский интерфейс на основе HTML API, с компоновкой информации из нераспределенных ограниченных контекстов на стороне сервера	743
Пример 2: Пользовательский интерфейс на основе DATA API, с компоновкой информации из распределенных ограниченных контекстов на стороне клиента	751
Ключевые идеи	758
Глава 24. CQRS: архитектура ограниченного контекста	760
Загружаемые примеры кода для этой главы	760
Проблемы использования единой модели для двух контекстов	761

Архитектура, более подходящая для сложных ограниченных контекстов	762
Команды: бизнес-задачи	763
Явное моделирование намерений	763
Модель, свободная от задач представления	765
Обслуживание бизнес-запросов	767
Запросы: информация о предметной модели	768
Отображение отчетов непосредственно в модель данных	769
Материализованные представления на основе предметных событий	769
Ошибочные представления о шаблоне CQRS	771
Шаблон CQRS сложен в реализации	771
Шаблон CQRS реализует потенциальную непротиворечивость	771
Модели должны поддерживать регистрацию событий	772
Команды должны выполняться асинхронно	772
Шаблон CQRS работает только с системами обмена сообщениями	772
Шаблон CQRS требует использовать предметные события	772
Шаблоны поддержки масштабируемости приложений	773
Масштабирование стороны чтения: потенциально непротиворечивая модель чтения	774
Масштабирование стороны записи: асинхронные команды	776
Масштабирование всего вместе	778
Ключевые идеи	778
Глава 25. Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования	780
Загружаемые примеры кода для этой главы	780
Различение прикладной и предметной логики	781
Прикладная логика	782
Предметная логика с точки зрения прикладной службы	793
Шаблоны прикладных служб	794
Процессор команд	794
Публикация/подписка	798
Шаблон запрос/ответ	800
async/await	802
Тестирование прикладных служб	803
Используйте предметную терминологию	803
Тестируйте как можно больше функциональных возможностей	805
Ключевые идеи	806
Глава 26. Запросы: предметная отчетность	807
Загружаемые примеры кода для этой главы	807
Предметная отчетность внутри ограниченного контекста	808
Создание отчетов на основе предметных объектов	808
Прямое обращение к хранилищу данных	815
Создание проекций из потоков событий	821
Предметная отчетность по нескольким ограниченным контекстам	829
С использованием приемов составных пользовательских интерфейсов	829
Отдельный контекст отчетов	830
Ключевые идеи	832

Моим дорогим крошкам — Примроуз и Альберту.

Скотт Миллетт

Об авторе

Скотт Миллетт (Scott Millett) — директор по информационным технологиям компании Iglu.com. Использует .NET, начиная с версии 1.0. В 2010 и 2011 годах был удостоен награды ASP.NET MVP Award. Является также автором книги «Professional ASP.NET Design Patterns and Professional Enterprise .NET». Если у вас появится желание обратиться к Скотту по вопросам, связанным с предметно-ориентированным проектированием или с работой в Iglu, вы можете написать ему по адресу scott@elbandit.co.uk, оставить сообщение в Twitter по адресу [@ScottMillett](https://twitter.com/ScottMillett) или подружиться в LinkedIn: <https://www.linkedin.com/in/scottmillett>.

О соавторе

Ник Тьюн (Nick Tune) со страстной увлеченностью решает задачи, возникающие перед бизнесом, создает впечатляющие программные продукты и непрестанно учится. Создание программного обеспечения — это работа его мечты. Пока что самым ярким этапом его карьеры была работа в 7digital, где он входил в состав самоорганизующихся команд, которые занимались решением бизнес-задач и разворачивали свои приложения на производственной площадке до 25 раз в день. Он рассчитывает на то, что в будущем его ждет работа над новыми захватывающими программными продуктами бок о бок с другими увлеченными людьми и возможность постоянно совершенствоваться в решении разнообразных задач.

Познакомиться с Ником ближе и больше узнать о его любимых технологиях и взглядах на разработку и распространение программного обеспечения можно на его веб-сайте (www.ntcoding.co.uk) и в Twitter ([@ntcoding](https://twitter.com/ntcoding)).

О научном редакторе

Энтони Деньер (Antony Denyer) выступает в роли разработчика, консультанта и коуча и профессионально занимается разработкой программного обеспечения с 2004 года. Он работал над целым рядом проектов, где эффективно применялись идеи и приемы предметно-ориентированного проектирования. Некоторое время назад он стал активно отстаивать использование CQRS и REST в большинстве своих проектов. Вы можете связаться с ним по электронной почте email@antonydenyer.co.uk или найти его в Twitter, где он пишет под именем [@tonydenyer](https://twitter.com/tonydenyer).

Благодарности

Прежде всего мне хотелось бы выразить огромную благодарность Нику Тьюну (Nick Tune) за его согласие помочь мне в работе над этим проектом и за огромный вклад во многие главы. Хочу сказать также спасибо Розмари Грэм (Rosemarie Graham), Джиму Майнтелу (Jim Minatel) и всем сотрудникам издательства Wrox, которые помогли мне в создании этой книги. Благодарю Энтони Деньера (Antony Denyer), безупречно выполнившего работу технического редактора. Наконец, большое спасибо Изабель Мак (Isabel Mack) за советы по правописанию и за замечания к черновой версии книги на Leanpub.

Введение

Писать программы легко — по крайней мере в том случае, когда речь идет о написании с нуля. Однако изменение существующего программного кода, который написали другие разработчики или даже вы сами, но месяцев, скажем, шесть назад, может в лучшем случае оказаться утомительным занятием, а в худшем — превратиться в суший кошмар. Программа работает, но вы не знаете точно, как именно. В ней есть все необходимые фреймворки, в ней применены правильные шаблоны проектирования, она создана с использованием приемов гибкой разработки — но добавить к коду новые возможности оказывается гораздо труднее, чем должно бы быть. Даже обращение к экспертам в предметной области ничего не дает, поскольку в коде не сохранилось никаких следов привычного для них языка. Работа над такой системой превращается в изнурительную расчистку, которая вызывает у разработчиков стойкое раздражение и лишает их всякого удовольствия от программирования.

Предметно-ориентированное проектирование¹ (Domain-Driven Design, DDD) — это процесс тесной увязки программного кода с реалиями предметной области. Благодаря ему добавление в программный продукт новых возможностей по мере его развития становится таким же простым, как при создании программы с нуля. Подход DDD не отрицает необходимости использовать шаблоны проектирования, принципы разработки, методологии и фреймворки, но ставит во главу угла сотрудничество разработчиков и экспертов в предметной области, которое позволяет достичь общего для всех понимания понятий, правил и логики предметной области. Полнота знаний о предметной области и тесный контакт с представителями бизнеса повышают шансы разработчиков создать программу, которая проще читается и легче адаптируется под будущие усовершенствования.

Следуя философии DDD, разработчики приобретают знания и навыки, необходимые для эффективной работы над большими и/или сложными программными системами для бизнеса. Будущие заявки на изменение системы не будут вызывать у них содрогание, а необходимость сопровождать и расширять унаследованный программный продукт перестанет служить поводом для уныния. Они начнут иначе смотреть даже на сам термин «унаследованный»: в их глазах унаследованная система станет системой, которая продолжает приносить пользу бизнесу.

Обзор книги и технологии

Из этой книги вы в подробностях узнаете о том, как применять шаблоны и приемы DDD в своих проектах. Однако перед детальным обсуждением полезно взгля-

¹ Можно также встретить название «проблемно-ориентированное проектирование». — *Примеч. пер.*

рованных областей. Благодаря изоляции неотъемлемых компонентов системы их последующие изменения не отразятся рикошетом на остальной системе.

В основе тех ключевых подсистем вашего продукта, которые достаточно сложны или будут часто изменяться, должна лежать модель. Тактические шаблоны DDD вкупе с архитектурой на основе модели (Model-Driven Design) помогут вам создать полезную модель предметной области в коде. Модель — это вместилище всей той прикладной логики, благодаря которой приложение выполняет все бизнес-сценарии использования. Модель отделена от технических сложностей, что оставляет простор для развития и изменения бизнес-правил и регламентов. Модель, согласованная с предметной областью, сделает ваше программное обеспечение легко адаптируемым и понятным для других разработчиков и специалистов со стороны бизнеса.

На рис. В.2 представлен общий вид пространства решений (solution space) DDD, о котором также рассказывается в первой части книги.

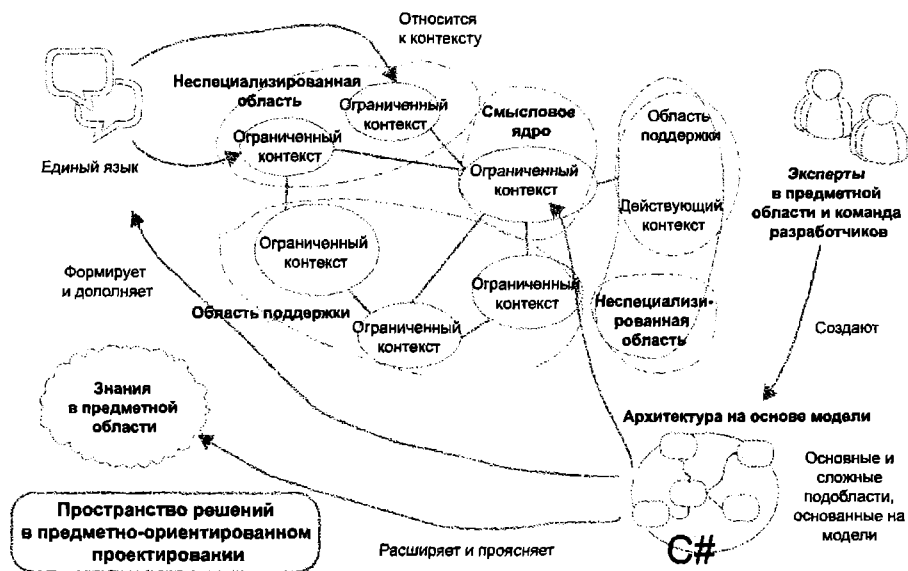


Рис. В.2. Общая схема пространства решений DDD

Структура этой книги

Эта книга делится на четыре части. Часть I посвящена философии, принципам и приемам предметно-ориентированного проектирования. В части II подробно обсуждаются стратегические шаблоны интеграции ограниченных контекстов. Часть III охватывает тактические шаблоны создания эффективных моделей предметной области. Часть IV в деталях описывает шаблоны проектирования, которые вы можете применять для того, чтобы извлекать пользу из модели предметной области и создавать эффективные приложения.

Часть I. Принципы и приемы предметно-ориентированного проектирования

Часть I знакомит вас с принципами и приемами DDD.

Глава 1. Что такое предметно-ориентированное проектирование?

DDD — это методология, помогающая справляться с проблемами, которые возникают при создании программного обеспечения для сложных предметных областей. Первая глава представляет философию DDD и рассказывает о том, почему терминология, сотрудничество и контекст являются самыми важными аспектами DDD и почему DDD — это гораздо больше, чем просто набор шаблонов программирования.

Глава 2. Дистилляция предметной области задачи

Освоение сложной предметной области чрезвычайно важно для создания удобного в сопровождении программного обеспечения. Процесс переработки знаний с участием экспертов в предметной области является ключом к этим знаниям. Глава 2 подробно рассказывает о приемах, которые позволяют команде разработчиков и экспертам в предметной области сотрудничать, совместно экспериментировать и обучаться, чтобы создавать эффективные модели предметной области.

Глава 3. Концентрация на смысловом ядре

В главе 3 рассказывается, как выполнить дистилляцию большой предметной области и выявить наиболее важную часть задачи — смысловое ядро (*core domain*). Затем мы объясняем, почему время и силы следует тратить прежде всего на смысловое ядро, изолировав его от менее важных областей поддержки (*supporting domains*) и неспециализированных областей (*generic domains*).

Глава 4. Проектирование на основе модели

Коллеги из бизнеса владеют аналитической моделью (*analysis model*) предметной области. У разработчиков есть собственная версия этой модели — программная модель (*code model*). Однако для успешного сотрудничества двух команд — разработчиков и специалистов от бизнеса — необходима единая модель. Единый язык (*ubiquitous language*) и разделяемое всеми представление о пространстве задачи — вот что связывает аналитическую модель с программной моделью. Идея единого языка является центральной для DDD и служит основой всей методологии. Единый язык для описания терминов и понятий предметной области, совместно выработанный командой разработчиков и бизнес-экспертами, крайне необходим для обсуждения сложных систем.

Глава 5. Шаблоны реализации предметной модели

Глава 5 представляет собой развернутый рассказ о том, за что отвечает и какую роль играет в приложении модель предметной области (domain model). Здесь представлены также различные шаблоны, которые можно применять при реализации модели предметной области, и описаны ситуации, для которых они являются наиболее подходящими.

Глава 6. Обеспечение целостности моделей предметной области с помощью ограниченных контекстов

В крупных решениях может существовать более одной модели. Важно сохранить целостность каждой модели, чтобы избежать неоднозначностей в языке и ненадлежащего использования понятий различными командами. Стратегический шаблон под названием «ограниченный контекст» (bounded context) создан специально для того, чтобы изолировать и защитить модель в контексте, обеспечив при этом возможность ее совместной работы с другими моделями.

Глава 7. Карты контекстов

Карта контекстов (context map), которая используется для того, чтобы понять и отразить отношения между разными моделями в приложении и способы их интеграции, является критически важным элементом стратегического проектирования. Карты контекстов охватывают не только технические аспекты интеграции, но и политические отношения между командами. Они дают общую картину ландшафта, которая помогает командам понять свою модель в контексте ландшафта в целом.

Глава 8. Архитектура приложения

Приложение должно уметь использовать модель предметной области, чтобы отвечать требованиям бизнес-сценариев использования. Глава 8 знакомит вас с архитектурными шаблонами, позволяющими структурировать приложение таким образом, чтобы сохранить целостность модели предметной области.

Глава 9. Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование

Глава 9 описывает характерные сложности, с которыми сталкиваются команды, применяющие DDD, и рассказывает, почему так важно знать, когда этот подход применять не следует. Здесь объясняется также, почему мощный инструментальный DDD в приложении к простым задачам может порождать избыточные и неоправданно усложненные системы.

Глава 10. Применение принципов, приемов и шаблонов DDD

Глава 10 повествует о том, как убедить других в преимуществах DDD и начать применять принципы и приемы этого подхода в своих проектах. Здесь объясняет-

ся, почему попытки создать идеальную модель предметной области не так ценны для создания качественного программного обеспечения, как исследования и эксперименты.

Часть II. Стратегические шаблоны: взаимодействие ограниченных контекстов

Часть II показывает, как интегрировать ограниченные контексты, и подробно описывает возможные подходы к конструированию ограниченных контекстов. Здесь представлены примеры кода, демонстрирующие тонкости интеграции с существующими приложениями, и описаны приемы, которые позволяют обеспечить коммуникацию через границы контекстов.

Глава 11. Введение в интеграцию ограниченных контекстов

Современные приложения — это распределенные системы, от которых требуются надежность и масштабируемость. Эта глава сводит воедино теорию распределенных систем и DDD, чтобы вы могли взять лучшее из обоих миров.

Глава 12. Интеграция посредством обмена сообщениями

Здесь мы в качестве иллюстрации создаем приложение, которое на примере организации шины сообщений для асинхронного обмена сообщениями показывает, как принципы построения распределенных систем применяются совместно с DDD.

Глава 13. Интеграция с RPC и REST посредством HTTP

Еще один пример приложения, демонстрирующий альтернативный подход к созданию асинхронных распределенных систем: вместо шины сообщений используются стандартные протоколы, такие как протоколы HTTP (протокол передачи гипертекста — Hypertext Transport Protocol), REST и Atom.

Часть III. Тактические шаблоны: создание эффективных моделей предметной области

Часть III охватывает шаблоны проектирования, которые можно использовать для построения модели предметной области в программном коде, наряду с шаблонами обеспечения сохранности модели и шаблонами управления жизненным циклом объектов предметной области, образующих модель.

Глава 14. Знакомство со стандартными блоками моделирования предметной области

Эта глава знакомит вас со всеми тактическими шаблонами, которые имеются в вашем распоряжении и позволяют создавать эффективные модели предметной

области. Приведенные здесь практические рекомендации помогают получать более управляемые и выразительные модели в программном коде.

Глава 15. Объекты-значения

Глава 15 служит введением в объекты-значения (value objects) — моделирующие конструкции DDD, которые позволяют представить такие лишённые индивидуальности понятия предметной области, как деньги.

Глава 16. Сущности

Сущности (entities) — это понятия предметной области, обладающие идентичностью, такие как клиенты, транзакции и отели. Эта глава содержит множество примеров и охватывает ряд дополнительных шаблонов реализации.

Глава 17. Службы предметной области

Некоторые понятия предметной области представляют собой операции без фиксации состояния, не относящиеся к каким-либо объектам-значениям или сущностям. Они известны как службы предметной области (domain services).

Глава 18. События предметной области

Во многих предметных областях можно достичь более глубокого понимания, если не ограничиваться исследованием одних лишь сущностей, а обратить пристальное внимание на события (events). В этой главе представлен шаблон проектирования на основе событий предметной области, который позволяет более ясно отражать события в модели предметной области.

Глава 19. Агрегаты

Агрегаты (aggregates) — это совокупности объектов, представляющих понятия предметной области. По сути дела, агрегаты ограничивают зону согласованности вокруг инвариантов. Это наиболее мощный из тактических шаблонов.

Глава 20. Фабрики

Фабрики (factories) — это шаблон организации жизненного цикла, который отделяет конструирование сложных объектов предметной области от их использования.

Глава 21. Репозитории

Репозитории (repositories) выступают посредниками между моделью предметной области и моделью данных. Они обеспечивают изоляцию модели предметной области от любых инфраструктурных особенностей и задач.

Глава 22. Регистрация событий

Регистрация событий (event sourcing), как и сами события, о которых рассказывается в главе 18, представляет собой полезный прием, который позволяет делать в программном коде акцент на событиях, возникающих в предметной области. Регистрация событий представляет собой шаг за пределы событий предметной области, поскольку сохраняет в виде событий состояние модели предметной области. Эта глава включает в себя целый ряд примеров, в том числе такие примеры, где используется специализированное хранилище событий.

Часть IV. Шаблоны проектирования эффективных приложений

В части IV представлены шаблоны для конструирования приложений, использующие модель предметной области и защищающие ее целостность.

Глава 23. Конструирование пользовательских интерфейсов приложения

Пользовательский интерфейс системы, состоящей из множества ограниченных контекстов, часто требует комбинирования данных из нескольких контекстов, особенно когда ограниченные контексты образуют распределенную систему.

Глава 24. CQRS: архитектура ограниченного контекста

CQRS (Command Query Responsibility Segregation — разделение ответственности на команды и запросы) — это шаблон проектирования, создающий две модели на месте одной. Там, где прежде была единая модель, обрабатывающая два разных контекста чтения и записи, создаются две отдельные модели — для обработки команд и для обслуживания запросов, на основе которых формируются отчеты.

Глава 25. Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования

Здесь описываются различия между прикладной и предметной логикой. Имея ясное представление об этих различиях, вы сумеете уберечь свою модель от разползания и упростить сопровождение системы.

Глава 26. Запросы: предметная отчетность

Чтобы принимать обдуманные решения в отношении разработки продукта и в деловой сфере, бизнес-специалистам нужна информация. В этой главе демонстрируются многочисленные приемы создания отчетов, обеспечивающих требуемый уровень информирования бизнеса.

Кому адресована эта книга

Основные темы DDD — приемы, шаблоны и принципы — обсуждаются здесь наряду с личным опытом и трактовкой методологии. Цель книги — помочь тем, кто заинтересовался этим подходом или только приступает к его использованию. Она не заменяет собой книгу «Предметно-ориентированное проектирование» Эрика Эванса¹, однако просто и доходчиво излагает концепции, введенные Эвансом, сопровождая описание практическими примерами, — с тем чтобы любой разработчик мог быстро войти в курс дела, прежде чем приступить к более глубокому изучению этой методологии.

В основе книги лежит личный опыт автора, полученный при использовании предмета обсуждения. Специалисты с большим опытом применения DDD могут в чем-то не согласиться с автором, но даже они, вероятно, смогут почерпнуть отсюда что-то полезное для себя.

Исходный код

Прорабатывая приведенные в книге примеры, вы можете вводить соответствующий программный код вручную или использовать сопроводительные файлы с исходным кодом. Весь исходный код примеров можно загрузить с сайта www.wrox.com. Ссылки для загрузки сопроводительных файлов конкретно к этой книге находятся на вкладке Download Code (Загрузить код) на странице: www.wrox.com/go/domaindrivendesign. Хотя все примеры написаны на языке C# для платформы .NET, предлагаемые концепции и приемы легко реализуются на любом другом языке программирования.

Чтобы найти на сайте www.wrox.com сопроводительные файлы с кодом к книге, можно пользоваться также поиском по значению ISBN (ISBN оригинала этой книги — 978-1-1187-1470-6). Полный список загружаемых примеров кода для книг, выпущенных издательством Wrox, доступен по адресу www.wrox.com/dynamic/books/download.aspx.

Опечатки

Мы приложили все усилия, чтобы не допустить ошибок в тексте и в программном коде. Однако никто не совершенен, и ошибки все же случаются. Если в какой-то из наших книг вы обнаружите ошибку — скажем, опечатку в тексте или дефектный фрагмент программного кода, — мы будем очень благодарны, если вы проинформируете нас об этом. Ваше сообщение поможет нам снабжать наших читателей еще более качественной информацией и, возможно, сэкономит силы и время кого-то из них.

¹ Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: ООО «И. Д. Вильямс», 2011. — *Примеч. пер.*

Чтобы увидеть текущий перечень ошибок, обнаруженных в этой книге, с поправками, подготовленными редакторами издательства Wrox, откройте страницу wrox.com/go/domaindrivendesign и перейдите по ссылке **Errata** (Ошибки).

Форумы P2P.WROX.COM

Чтобы обсудить эту книгу с автором и коллегами, зарегистрируйтесь на дискуссионной площадке P2P по адресу <http://p2p.wrox.com>. Эта онлайн-дискуссионная площадка создана для читателей книг издательства Wrox и пользователей технологий, описанных в этих книгах: здесь они могут обмениваться мнениями и взаимодействовать друг с другом. Инструментарий площадки позволяет подписаться на обновления интересующих вас тем, чтобы получать уведомления о новых сообщениях по электронной почте. В работе площадки принимают участие авторы книг, редакторы издательства Wrox, специалисты из разных отраслей, а также читатели.

На страницах площадки <http://p2p.wrox.com> вы найдете множество разных дискуссий, которые помогут вам не только при чтении этой книги, но и при разработке собственных приложений. Процедура регистрации проста:

1. Откройте страницу <http://p2p.wrox.com> и перейдите по ссылке **Register** (Зарегистрироваться).
2. Ознакомьтесь с условиями использования форумов, поставьте флажок **I have read, and agree to abide by the p2p.wrox.com Forums rules** (Я ознакомился с правилами форумов p2p.wrox.com и обязуюсь соблюдать их) и щелкните на кнопке **Register** (Зарегистрироваться).
3. Заполните обязательные поля предложенной формы и (при желании) необязательные, а затем щелкните по кнопке **Complete Registration** (Завершить регистрацию).
4. После этого на указанный адрес электронной почты будут высланы инструкции, описывающие, как подтвердить учетную информацию и завершить процедуру регистрации.

ПРИМЕЧАНИЕ

Чтение сообщений на форумах не требует регистрации, но для публикации своих сообщений регистрация необходима.

После регистрации вы сможете оставлять на форумах сообщения и отвечать на сообщения других пользователей. Вы можете в любое время читать форумы онлайн, а если хотите, чтобы новые сообщения из определенного форума поступали к вам по электронной почте, щелкните на кнопке **Subscribe** (Подписаться) рядом с именем форума в списке.

Дополнительные сведения о том, как пользоваться форумами Wrox P2P, вы найдете в сборнике ответов на часто задаваемые вопросы P2P FAQ. Здесь можно уз-

нать также о том, как работает программное обеспечение дискуссионной площадки P2P, и получить востребованную информацию о самой площадке и о книгах издательства Wrox. Чтобы перейти к сборнику ответов на часто задаваемые вопросы, щелкните на ссылке FAQ на любой странице форумов P2P.

Резюме

Цель этой книги — представить методологию DDD в доступной и практической форме в расчете на опытных разработчиков, которые занимаются созданием сложных приложений. Основное внимание здесь уделено принципам и приемам разделения сложного пространства задачи на более простые части, а также шаблонам реализации и удачным приемам формирования управляемого пространства решений. Из этой книги вы узнаете, как конструировать эффективные модели предметной области с помощью тактических шаблонов и как обеспечивать целостность этих моделей, используя стратегические шаблоны DDD.

К концу книги вы получите ясное представление о философии DDD. Вы сможете убедительно рассказать другим о преимуществах этой методологии и о том, когда следует ее применять. Вы поймете, что при всей полезности тактических шаблонов DDD конструировать масштабируемые и простые в сопровождении приложения помогают прежде всего принципы, приемы и стратегические шаблоны. Овладев сведениями, представленными в этой книге, вы сможете успешнее справиться с созданием сложного программного обеспечения для обширных и сложно устроенных предметных областей.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

Принципы и приемы предметно- ориентированного проектирования

Глава 1: Что такое предметно-ориентированное проектирование?

Глава 2: Дистилляция предметной области задачи

Глава 3: Концентрация на смысловом ядре

Глава 4: Проектирование на основе модели

Глава 5: Шаблоны реализации предметной модели

Глава 6: Обеспечение целостности моделей предметной области с помощью ограниченных контекстов

Глава 7: Карты контекстов

Глава 8: Архитектура приложения

Глава 9: Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование

Глава 10: Применение принципов, приемов и шаблонов DDD

1

Что такое предметно-ориентированное проектирование?

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в философию предметно-ориентированного проектирования
- Проблемы разработки программного обеспечения для сложных предметных областей
- Как предметно-ориентированное проектирование помогает управлять сложностью
- Как предметно-ориентированное проектирование применяется к пространству задачи и пространству решения
- Стратегические и тактические шаблоны предметно-ориентированного проектирования
- Приемы и принципы предметно-ориентированного проектирования
- Ошибочные представления о предметно-ориентированном проектировании

Предметно-ориентированное проектирование (DDD) — это методология разработки, определение которой дал Эрик Эванс в своей книге «Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем»¹. DDD представляет собой подход к разработке программного обеспечения, позволяющий коллективам эффективно управлять конструированием и поддержкой программного обеспечения для сложных предметных областей.

Эта глава дает общее представление о приемах, шаблонах и принципах DDD, а также рассказывает о том, как эта методология позволит усовершенствовать ваш подход к разработке программного обеспечения. Вы узнаете, в чем заключается ценность анализа пространства задачи и на чем следует сосредоточить свои усилия в первую очередь. Вы поймете, почему сотрудничество, коммуникация и контекст настолько важны для проектирования удобного и простого в сопровождении программного обеспечения.

¹ Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: ООО «И. Д. Вильямс», 2011. — *Примеч. пер.*

К концу главы у вас сформируется ясное понимание DDD, которое послужит основой для исследования тонкостей разных шаблонов, приемов и принципов, описанных в этой книге. Однако прежде чем углубиться в особенности управления сложностью с помощью DDD, важно разобраться в том, почему программное обеспечение может прийти в неуправляемое состояние.

Проблемы разработки программного обеспечения для сложных предметных областей

Чтобы понять, чем методология DDD полезна в проектировании программного обеспечения для нетривиальных предметных областей, нужно сначала выяснить, какие сложности возникают при создании и сопровождении программ. Без сомнения, самый популярный шаблон проектирования архитектуры прикладных программ — это «Большой ком грязи» (Big Ball of Mud, BBoM). Определение BBoM дали Брайан Фут (Brian Foote) и Йозеф Йодер (Joseph Yoder) в статье «Big Ball of Mud»: «...лишенный видимой структуры, беспорядочный, рыхлый код, напоминающий комок спутанной проволоки или спагетти».

Термином BBoM Фут и Йодер описали приложение, у которого нет различимой архитектуры (представьте себе большую тарелку спагетти рядом с лазаньей, уложенной аккуратными слоями). Превращение программы в «ком грязи» становится явным, когда реализация даже простых изменений в алгоритме работы и небольших улучшений наталкивается на то, что существующий программный код трудно читать и понимать. В своей книге Эрик Эванс описывает такие системы как «механизм, который делает что-то полезное, но ничего не объясняет». Одна из главных причин того, что программы становятся сложными и неуправляемыми, заключается в наложении сложности предметной области на техническую сложность, как показано на рис. 1.1.

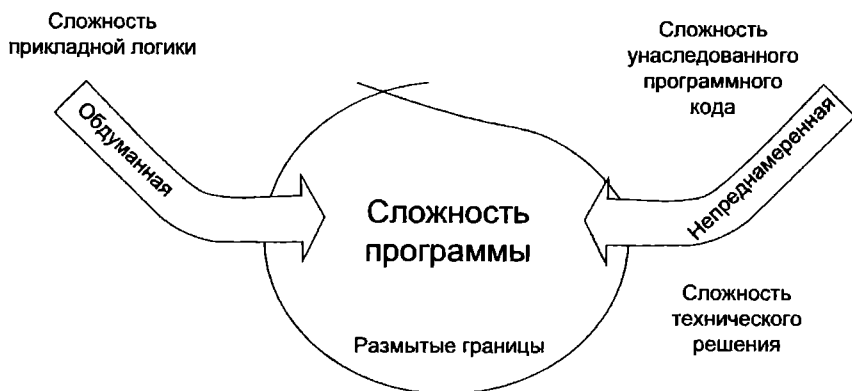


Рис. 1.1. Сложность программного обеспечения

Код, созданный без использования общего языка

Недостаточное внимание к общему языку и знаниям в предметной области порождает программный код, который вроде бы работает, но не раскрывает как следует намерения бизнеса. Это делает код трудным для чтения и сопровождения, поскольку необходимость постоянно переводить понятия между аналитической и программной моделями усложняет работу и служит источником ошибок.

Без увязки с аналитической моделью, которую понимают бизнес-специалисты, код будет со временем ухудшаться и в конце концов скатится до архитектуры, напоминающей шаблон ВВоМ. Из-за затрат на подобный перевод команды избегают использовать в коде обширный лексикон предметной области и тем самым сокращают свои шансы обнаружить новые понятия предметной области в ходе сотрудничества с бизнес-специалистами.

ЧТО ТАКОЕ АНАЛИТИЧЕСКАЯ МОДЕЛЬ?

Аналитическая модель используется для описания логической организации и структуры прикладной программы. Она может быть представлена в виде схем или с помощью языков моделирования, таких как UML. Это представление программы, которое помогает неспециалистам понять, как сконструирована программа.

Недостаточная организация

Как показывает рис. 1.2, первое воплощение системы, напоминающее шаблон ВВоМ, не требует много времени и даже зачастую имеет успех, но из-за недостаточного внимания к модели предметной области при проектировании приложе-

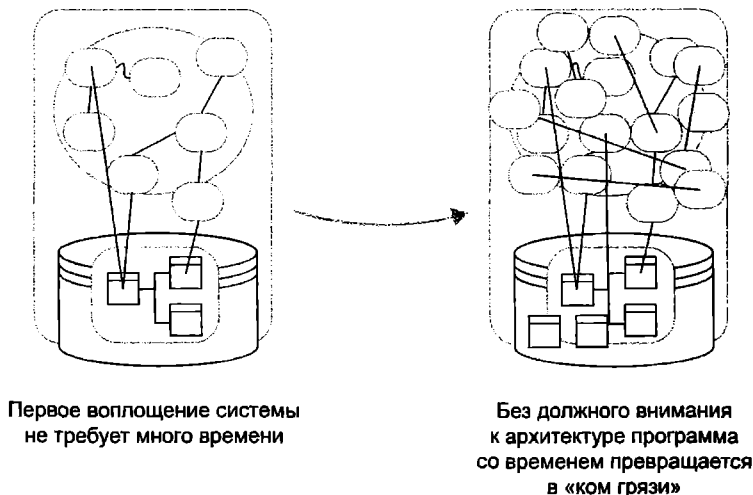


Рис. 1.2. Распад кода

ния последующие расширения даются с большим трудом. Программному коду не хватает синергии с предметной областью и бизнес-поведением системы, из-за чего становится сложно управлять изменениями. Сложность предметной области часто накладывается на непреднамеренную сложность технического решения.

Шаблон «Большой ком грязи» тормозит развитие продукта

Сохранение спагетти-подобной архитектуры часто замедляет добавление в продукт новых функций. Последующие версии программного продукта начинают кишеть программными ошибками, поскольку разработчикам приходится иметь дело с малопонятной «кашей» кода. С течением времени разработчики все чаще жалуются на то, что с кодом сложно работать. Даже вливание в проект дополнительных ресурсов не дает той скорости разработки, которая устроила бы бизнес-заказчика.

В конце концов ситуация накаляется до такой степени, что встает вопрос о полной переделке приложения. Однако без должного внимания к вопросам проектирования новый проект столкнется с теми же самыми проблемами, которые породили исходный «ком грязи». Все это в конечном итоге может разочаровать бизнес-заказчика: ведь хотя поначалу вложения приносят зримую высокую отдачу в терминах функциональности и скорости разработки, затем даже с ростом инвестиций развитие продукта замедляется настолько, что перестает соответствовать требованиям бизнеса. В целом шаблон ВВоМ представляет собой весьма серьезную проблему и для вас как разработчика, поскольку вам едва ли понравится работать с этой жуткой мешаниной кода, и для заказчика, поскольку ставит крест на возможностях быстро принести пользу бизнесу.

ЧТО ТАКОЕ ПРЕДМЕТНАЯ ОБЛАСТЬ?

Под предметной областью понимается область деятельности, для которой вы создаете программное обеспечение. Философия DDD подчеркивает необходимость уделять предметной области первоочередное внимание при работе над программным обеспечением для крупномасштабных и сложных бизнес-задач. Эксперты в предметной области помогают разработчикам сконцентрироваться на тех моментах своей области деятельности, которые наиболее значимы для создания полезного программного продукта. Например, чтобы написать программное обеспечение для сферы здравоохранения, позволяющее вести электронные карты пациентов, не нужно становиться врачом, но важно разбираться в терминологии медицинских работников и понимать, какая информация о пациентах интересует разные отделы, какие сведения собирает врач и как все это используется.

Недостаточное внимание к предметной области

Когда разработчики недостаточно хорошо понимают предметную область, программный проект обречен на провал. Кодирование как таковое не самое узкое место в процессе создания продукта; это самая простая часть процесса разработки.

Гораздо сложнее создать и поддерживать полезную программную модель предметной области, способную в полной мере отвечать бизнес-сценариям использования. Однако чем больше сил вы потратите на то, чтобы разобраться в предметной области, тем лучше будете готовы, когда придет время воплотить эту модель в программном коде для решения бизнес-задач.

Как шаблоны предметно-ориентированного проектирования помогают обуздать сложность

Подход DDD помогает не только разобраться в самой предметной области, но и создать продукт, полезный для решения задач в этой области. Это достигается использованием ряда стратегических и тактических шаблонов.

Стратегические шаблоны DDD

Стратегические шаблоны DDD вычленяют суть предметной области и формируют архитектуру приложения.

Выявление наиболее важных аспектов с помощью дистилляции¹ предметной области

Не все элементы большого программного продукта нуждаются в идеальном проектировании — порой это становится пустой тратой времени. С помощью аналитических шаблонов и собранных знаний разработчики и эксперты в предметной области преобразуют обширную предметную область в набор более мелких подобластей. Такое преобразование помогает выявить смысловое ядро — то, ради чего создается продукт. Смысловое ядро служит движущей силой разработки продукта, фундаментом для его создания. DDD подчеркивает, что силы и навыки команды надо направить прежде всего на смысловое ядро, так как именно эта область наиболее значима и определяет успешность продукта.

Ясно понимая, что требует первоочередного внимания, а что вторично, разработчики могут попробовать найти готовые решения с открытым кодом для реализации некоторых менее важных элементов системы, и у них останется больше времени на то, что действительно важно. Это поможет предотвратить превращение смыслового ядра в «ком грязи».

Вычленение смыслового ядра помогает разработчикам понять, для чего создается продукт и от чего зависит его успех с точки зрения бизнес-заказчика. Именно правильное представление о намерениях бизнеса позволяет разработчикам выявить

¹ В DDD под дистилляцией понимается выделение сути, концентрация внимания на наиболее важных аспектах задачи. Термин «дистилляция» употребляется здесь по аналогии с химией, где дистилляцией называют процесс разделения компонентов смеси с целью выделения основного вещества в такой форме, которая делает его более ценным и полезным. — *Примеч. пер.*

самые важные части системы и заняться прежде всего ими. Вместе с развитием бизнеса должен развиваться и программный продукт, а для этого он должен обладать достаточной гибкостью. Благодаря повышенному вниманию к качеству кода ключевых частей приложения впоследствии будет легко адаптировать продукт к изменяющимся потребностям бизнеса. Если же ключевые части продукта будут плохо согласованы с предметной областью, то со временем архитектура приложения рассыплется и превратится в «ком грязи», что серьезно затруднит сопровождение.

Создание модели для решения задач предметной области

В пространстве решения для каждой предметной подобласти создается своя программная модель, которая обслуживает решение соответствующих предметных задач и вписывает продукт в контуры бизнеса. Это не столько фактическая модель, сколько абстракция, благодаря которой продукт реализует требования бизнес-сценариев использования, поддерживая правила и логику предметной области. На создание модели и прикладной логики разработчики должны направить столько же сил и энергии, сколько на чисто технические аспекты приложения. Чтобы избавиться от непреднамеренной технической сложности, модель следует изолировать от инфраструктурного кода.

Модели отличаются друг от друга; вместо какого-то общего шаблона проектирования для каждой модели выбирается свой наиболее подходящий шаблон, учитывающий уровень сложности соответствующей подобласти. В не очень сложных или не существенных для успеха продукта подобластях нет необходимости прибегать к объектно-ориентированному проектированию — можно обойтись процедурной архитектурой или архитектурой, управляемой данными.

Использование общего языка для совместной работы над моделями

Разработчики и эксперты в предметной области строят модели в тесном сотрудничестве. Для общения друг с другом они используют постоянно эволюционирующий общий язык, который называется единым языком. Он помогает эффективно увязывать между собой программную модель и концептуальную аналитическую модель. Эта увязка достигается использованием одних и тех же терминов в структуре модели и архитектуре классов. Внутренние представления, понятия и термины, выработанные на уровне программного кода, переносятся в единый язык и, соответственно, в аналитическую модель. Встречным образом понятия, которые выявляются на уровне аналитической модели, переносятся в программную модель. Это позволяет экспертам в предметной области и разработчикам совместно развивать модель.

Изоляция модели как способ избежать неоднозначностей и искажений

Модели заключены в ограниченный контекст, который определяет применимость модели и гарантирует ее целостность. Если в терминологии возникла неоднозначность или над проектом работают несколько команд, для сокращения сложности

можно разбивать большие модели на более мелкие, помещая их в отдельные ограниченные контексты.

Ограниченные контексты позволяют сформировать вокруг моделей защитные границы, которые не дают продукту превратиться в «ком грязи». Это достигается путем создания для общего решения разных моделей, развивающихся внутри четко определенных предметных контекстов и не оказывающих негативного влияния на другие части системы. Модели изолируются от инфраструктурного кода, чтобы избежать возникновения непредвиденной сложности при смешении технических и предметных понятий. Ограниченные контексты также защищают целостность моделей, изолируя их от стороннего программного кода.

Сравните диаграммы на рис. 1.2 и рис. 1.3. Диаграмма на рис. 1.3 показывает, как стратегические шаблоны DDD применяются для управления большой предметной областью и для защиты отдельных моделей внутри нее.

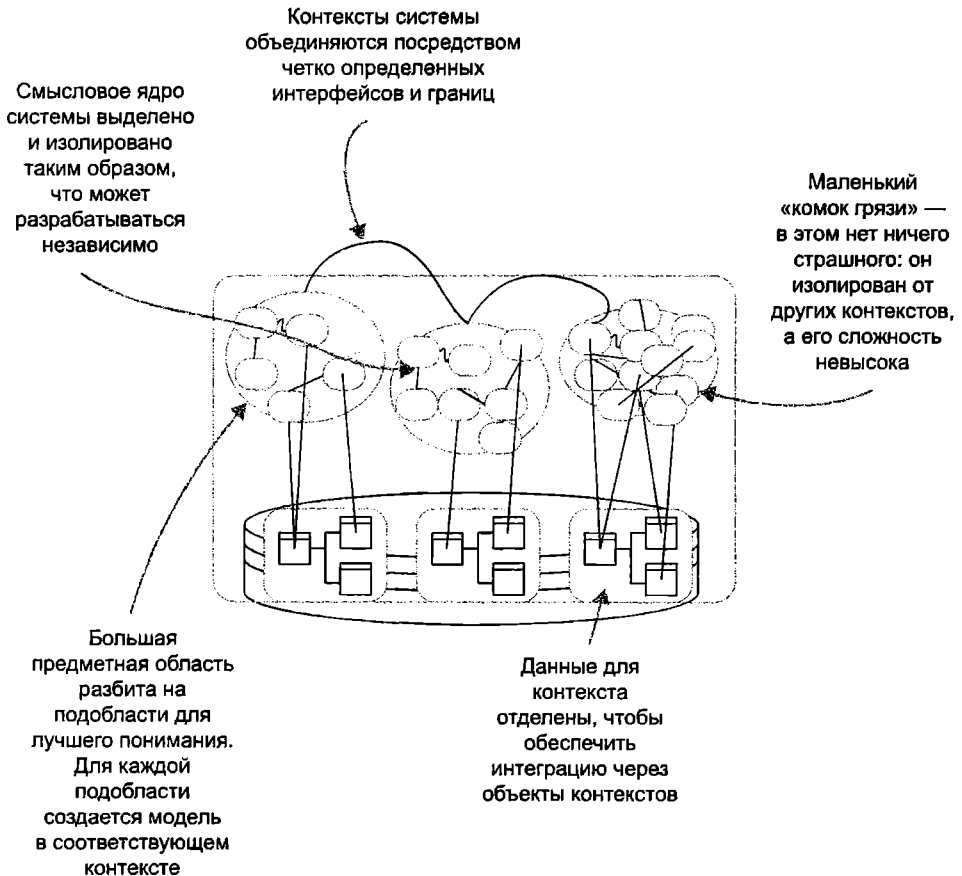


Рис. 1.3. Применение стратегических шаблонов предметно-ориентированного проектирования

«КОМ ГЯЗИ» НЕ ВСЕГДА ЯВЛЯЕТСЯ АНТИШАБЛОНОМ

Не все части большого приложения спроектированы идеально — но это и не нужно. Хотя строить весь программный стек предприятия на основе шаблона ВВоМ категорически не рекомендуется, этот шаблон все же можно применять на практике. Области, сложность которых невысока или которые наверняка не получат дальнейшего развития, могут конструироваться без предъявления высоких требований к качеству кода; для них вполне достаточно просто получить работоспособный код. Иногда для успеха продукта критически важными оказываются первенство на рынке и быстрое получение обратной связи; тогда может оказаться целесообразным получить работающую программу как можно скорее, не уделяя особого внимания ее архитектуре. Качество кода всегда можно улучшить позже, когда бизнес придет к выводу, что продукт пользуется успехом и есть смысл продолжить инвестиции в его производство и развитие. Ключом к успеху применения ВВоМ является определение контекста вокруг ограниченных контекстов, использующих ВВоМ, чтобы уберечь от повреждения смысловое ядро.

Понимание отношений между контекстами

Важно добиться, чтобы разработчики и эксперты в предметной области ясно понимали, каким образом отдельные модели и контексты взаимодействуют при решении задач, затрагивающих несколько подобластей. Общую картину в DDD помогают охватить карты контекстов; благодаря им разработчики видят, какие модели существуют, за что они отвечают и где проходят границы их применимости. Эти карты показывают, как разные модели взаимодействуют и какими данными они обмениваются, поддерживая те или иные бизнес-процессы. Эксперты в предметной области часто не отслеживают и не очень хорошо понимают связи моделей друг с другом и, что еще более важно, с «ничейными» областями процесса между ними.

Тактические шаблоны DDD

Тактические шаблоны DDD, также известные как строительные блоки моделей, — это набор шаблонов, помогающих создавать эффективные модели для сложных ограниченных контекстов. Многие шаблоны проектирования, помещенные в этот набор, широко использовались еще до появления работы Эванса и были классифицированы Мартином Фаулером в книге «Архитектура корпоративных программных приложений»¹ и Эрихом Гаммой с соавторами в книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования»². Эти шаблоны применимы не ко все моделям, и каждый из них следует использовать с учетом его собственных достоинств и в рамках определенного архитектурного стиля.

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: ООО «И. Д. Вильямс», 2010. (Ранее выходила в том же издательстве под названием «Архитектура корпоративных программных приложений».) — *Примеч. пер.*

² Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2016. — *Примеч. пер.*

Пространство задачи и пространство решения

Все шаблоны, упоминаемые в этом разделе, помогают управлять сложностью задачи (пространства задачи) или сложностью решения (пространства решения). Пространство задачи, как показано на рис. 1.4, преобразует предметную область в набор более управляемых подобластей. Цель подхода DDD в применении к пространству задачи — выделить наиболее важные области, на которых необходимо сосредоточить внимание. В следующей главе мы подробнее рассмотрим шаблоны, помогающие снизить сложность пространства задачи.

Пространство решения DDD, изображенное на рис. 1.5, охватывает шаблоны, помогающие сформировать архитектуру приложения и сделать ее более управляемой.

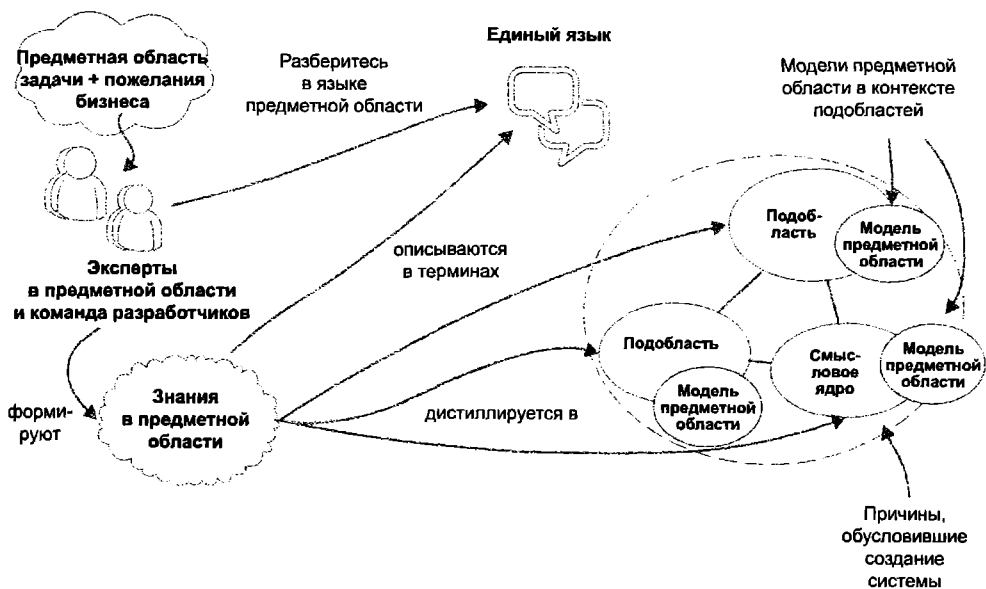


Рис. 1.4. Шаблоны DDD, применимые к пространству задачи

Приемы и принципы предметно-ориентированного проектирования

Помимо множества шаблонов, в DDD имеется ряд приемов и руководящих принципов, в значительной мере обеспечивающих успех при использовании этого подхода. Эти ключевые принципы, составляющие суть DDD, часто упускают из виду, уделяя основное внимание тактическим шаблонам проектирования, которые используются для создания программных моделей.

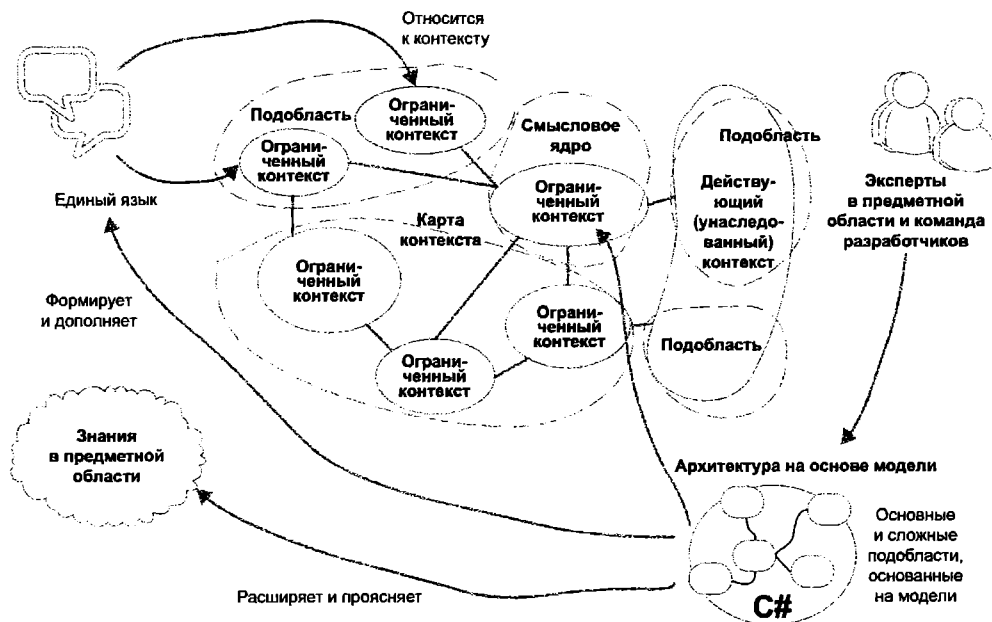


Рис. 1.5. Шаблоны DDD, применимые к пространству решения

Концентрация на смысловом ядре

DDD подчеркивает, что ваши усилия должны быть сосредоточены на смысловом ядре. Смысловое ядро — это та часть вашего продукта, от которой зависит его успех или провал. Это то уникальное свойство, которое служит основной причиной создания, а не покупки программного продукта. Смысловое ядро — это та часть продукта, которая обеспечивает его конкурентные преимущества и формирует его истинную ценность для бизнеса. Очень важно, чтобы все члены команды понимали, что такое смысловое ядро.

Изучение через сотрудничество

DDD подчеркивает важность совместной работы разработчиков и экспертов в предметной области над созданием удачных моделей для решения задач. Если эксперты в предметной области не готовы к такому сотрудничеству, обмен знаниями не состоится — и разработчики не смогут глубоко вникнуть в суть предметной области. Верно и то, что благодаря сотрудничеству и формированию общего свода знаний коллеги из бизнеса и сами получают возможность гораздо больше узнать о своей предметной области.

Создание моделей посредством исследований и экспериментов

DDD интерпретирует аналитическую и программную модели как единое целое, в котором программная модель связана с аналитической посредством единого языка. Подвижки в аналитической модели ведут к изменению программной модели. Рефакторинг программной модели, выявляющий более глубокие слои содержания задачи, в свою очередь, отражается в аналитической и ментальной модели бизнеса. Подвижки и усовершенствования становятся возможными тогда, когда разработчикам выделяется время на исследование модели и эксперименты с ее архитектурой. Моделирование и эксперименты важны для выбора лучшей архитектуры. В ходе исследований и экспериментов можно также понять, какие модели неудачны. Эрик Эванс считает, что на каждую удачную архитектуру должно приходиться не меньше трех неудачных: это стимулирует разработчиков не останавливаться на первой же работоспособной модели.

Коммуникация

Способность эффективно описать модель, представляющую предметную область, является основой DDD. Вот почему наиважнейшим аспектом DDD является создание единого языка. Без общего языка сотрудничество бизнеса с разработчиками в решении задач будет малоэффективным. Общий язык необходим для увязки технической реализации с аналитической и ментальной моделями, которые получены при формировании общего свода знаний. Наличие эффективного способа выражать идеи и решения из предметной области делает возможным усовершенствование архитектуры.

Именно сотрудничество и конструирование единого языка — источник мощи DDD. Благодаря им и бизнес-специалисты, и разработчики глубже понимают предметную область и эффективнее взаимодействуют друг с другом. Эти ключевые аспекты оказывают огромное влияние на проект. Хотя базовая архитектура и методологии, безусловно, важны, подход DDD в той же (если не в большей) мере опирается на анализ и понимание предметной области, что в конечном счете обеспечивает успех программного продукта.

Понимание границ применимости модели

Каждая построенная модель осмысливается в контексте конкретной подобласти и описывается с использованием единого языка. Однако при работе с большими моделями в едином языке могут возникать неоднозначности, когда в разных подразделениях организации по-разному понимают одни и те же общие термины или понятия. В DDD эта проблема решается созданием для каждой модели собственного единого языка, действительного только в определенном контексте. Каждый контекст задает языковые границы, а привязка модели к определенному контексту позволяет избежать терминологических неоднозначностей в данном контексте. Соответственно модель, в которой возникла терминологическая двусмысленность, следует разделить на две части, каждая из которых должна быть определена

в собственном контексте. С точки зрения реализации такие языковые границы обеспечиваются стратегическими шаблонами, что дает моделям возможность развиваться независимо. Такие стратегические шаблоны порождают хорошо организованный программный код, который легко сопровождать и изменять.

Постоянное развитие модели

Любой разработчик, работающий над сложной системой, может написать качественный код и какое-то время сопровождать его. Однако без тесной связи исходного кода с предметной областью продолжительная разработка, весьма вероятно, приведет к появлению программного кода, который сложно изменять и который склонен со временем превращаться в «ком грязи». DDD помогает решить эту проблему, призывая разработчиков постоянно оценивать, насколько модель соответствует текущей задаче, развивать сложные модели по мере проникновения в суть предметной области и упрощать их. DDD не панацея и требует постоянной переработки и расширения полученных знаний, чтобы программный продукт на выходе можно было сопровождать годы, а не месяцы. Появление новых бизнес-сценариев способно превратить прежде удачную модель в неудачную или потребовать изменений, которые более четко отражают новые или существующие концепции.

Типичные заблуждения, связанные с предметно-ориентированным проектированием

Предметно-ориентированное проектирование (DDD) можно рассматривать как философию разработки; эта философия пропагандирует новый способ мышления, ориентированный на предметную область. Однако самой сильной стороной DDD является вовсе не стремление к какому-то конечному результату, а процесс обучения сам по себе. Любой коллектив разработчиков сможет написать программный продукт, отвечающий заданному набору вариантов использования, но только команда, которая тратит время и силы на изучение той предметной области, где ведется разработка, сумеет непрерывно развивать и совершенствовать свой продукт, отзываясь на все новые потребности бизнеса. По своей природе DDD не является строгой методологией; для создания и дальнейшего развития полезной модели необходимо использовать ее в сочетании с некоторой другой методологией итеративного проектирования программного продукта.

Суть DDD — в тактических шаблонах

DDD не является ни учебником объектно-ориентированного проектирования, ни философией написания программного кода, ни языком шаблонов. Однако поиск статей о DDD в Интернете может создать ложное впечатление, что это всего лишь горстка шаблонов реализации, поскольку подавляющее большинство статей и блогов, посвященных DDD, фокусируются именно на шаблонах моделиро-

вания. Разработчикам намного проще копать в том, как тактические шаблоны DDD реализуются в программном коде, нежели общаться по поводу предметной области, в которой они не разбираются и которая их не заботит, с пользователями и специалистами из бизнеса. Именно поэтому в DDD иногда ошибочно видят просто язык шаблонов, составленный из сущностей, объектов-значений и хранилищ. На деле DDD можно применять вовсе без создания детальной модели предметной области или использования хранилища. Однако методология DDD ориентирована не столько на шаблоны проектирования программного обеспечения, сколько на совместное решение задач.

Эванс показал, каким образом с помощью шаблонов проектирования разработчики и бизнес-специалисты могут создавать модели, реализуемые с использованием единого языка. Однако сама по себе программная реализация мало что значит без методов анализа и тесного сотрудничества. DDD не акцентируется на программной стороне задачи; цель этой методологии не в том, чтобы написать красивый код. Программный код — это лишь один из артефактов DDD.

DDD — это фреймворк

DDD не требует использования специализированного фреймворка или базы данных. Модель воплощается в коде в соответствии с принципом POCO (Plain Old C# Object — старый добрый объект C#), который избавляет от любого инфраструктурного кода, позволяя разработчику полностью сосредоточиться на предметной области. Объектно-ориентированная методология полезна для конструирования моделей, но не является обязательной.

DDD безразлична к архитектуре в том смысле, что не требует следовать какому-то определенному архитектурному стилю. Эванс в своей книге представляет многоуровневую архитектуру, но этот стиль не является единственно возможным. Архитектурные стили должны применяться на уровне ограниченного контекста, а не на уровне приложения, а потому могут различаться. В рамках одного и того же программного продукта один ограниченный контекст может быть основан на событиях, другой — использовать многоуровневую предметную модель, а третий — применять шаблон Active Record.

DDD — это панацея

Подход DDD может отнимать много сил, он подразумевает использование методологии итеративной разработки, вынуждает активно подключать к процессу коллег из бизнеса и требует толковых разработчиков. Использование приемов анализа DDD (таких, как дистилляция предметной области) и стратегических шаблонов (таких, как изоляция программной модели, представляющей прикладную логику) может принести пользу любому программному проекту. Однако отнюдь не любому проекту для построения подробной модели предметной области нужны все тактические шаблоны DDD. В простейших предметных областях такая изощренность неоправдана, поскольку прикладная логика проста или практически отсутствует. Например, попытка применить все шаблоны DDD при создании простого приложения для управления блогами будет напрасной тратой времени и сил.

Ключевые идеи

- Предметно-ориентированное проектирование (DDD) — это философия разработки, цель которой — помочь справиться с созданием и сопровождением программного обеспечения для сложных предметных областей.
- DDD — это набор шаблонов, принципов и приемов, которые могут использоваться при проектировании программного обеспечения для управления его сложностью.
- DDD включает в себя шаблоны двух типов. Стратегические шаблоны формируют решение, тогда как тактические шаблоны используются для реализации предметной модели. Стратегические шаблоны могут принести пользу любому приложению, тактические же полезны лишь там, где модель содержит достаточно изощренную прикладную логику.
- Дистилляция большой предметной области в несколько подобластей помогает выделить смысловое ядро — наиболее ценную часть системы. Не все части системы требуют безупречного проектирования — основное внимание разработчики должны уделять смысловому ядру (или ядрам) продукта.
- Для управления задачами предметной области в каждой подобласти создается абстрактная модель.
- Чтобы разработчики и эксперты могли совместно проектировать модель, используется единый язык, связывающий аналитическую и программную модели друг с другом. Изучение и создание языка для обсуждения предметной области — это процесс DDD. Программный код — лишь артефакт.
- Для сохранения целостности модели она определяется в ограниченном контексте. Модель изолируется от инфраструктурных особенностей приложения, чтобы отделить техническую сложность задачи от сложности предметной области.
- Когда в терминологии модели появляются неоднозначности или над моделью работают несколько команд, модель можно разбить на несколько более мелких, определив более узкие ограниченные контексты.
- Подход DDD не обязывает применять определенный архитектурный стиль разработки, он лишь добивается изоляции модели от технических сложностей, чтобы разработчики могли сосредоточиться на прикладной логике.
- DDD придает большое значение концентрации на смысловом ядре, сотрудничеству, совместному исследованию с экспертами в предметной области, экспериментированию в поисках наиболее удачной модели и освоению разных контекстов, составляющих сложную предметную область.
- DDD — это не язык шаблонов, а философия сотрудничества, которая помещает в центр внимания ценность поставляемого продукта и отводит ключевую роль коммуникации.
- DDD — это подход к разработке программного обеспечения, сосредоточенный на предметной области задачи и направленный на выработку единого языка предметной области.

2

Дистилляция предметной области задачи

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Зачем нужна переработка знаний
- Как сотрудничество формирует единые представления и общий язык
- Кто такой «эксперт в предметной области» и почему его роль так важна
- Эффективные методы приобретения знаний о предметной области

Понимание предметной области, необходимое для создания простой и практичной модели, требует обширных и глубоких познаний, которые можно приобрести только через совместную работу с людьми, владеющими этой областью от и до. Мощь DDD заключена в непрерывном экспериментировании и исследованиях в ходе проектирования модели. Только через сотрудничество и выработку общих представлений о предметной области можно эффективно спроектировать такую модель для решения задач, стоящих перед бизнесом, которая была бы достаточно гибкой, чтобы адаптироваться по мере появления новых требований.

В этой главе рассказывается о методах, которые облегчают дистилляцию знаний в предметной области. Благодаря им вы сможете лучше разобраться в предметной области задачи, что позволит вам создавать более эффективные предметные модели. Здесь будут представлены также методы извлечения важной информации о поведении приложения и приемы, которые помогают глубже вникнуть в суть предметной области.

Переработка знаний и сотрудничество

Сложные предметные области содержат массу информации, часть которой не имеет отношения к решению рассматриваемой задачи и будет лишь отвлекать вас от истинной цели ваших усилий по моделированию. Переработка¹ знаний

¹ Слово «переработка» тут следует понимать так же, как в словосочетании «перерабатывающая промышленность»: это процесс, посредством которого из набора знаний, существу-

(knowledge crunching) — это искусство выделения из предметной области той информации, которая связана с сутью задачи, с тем чтобы построить полезную модель, отвечающую требованиям бизнес-сценариев использования.

Переработка знаний помогает заполнить пробелы в познаниях разработчиков при разработке решения для предметной области на основе набора требований. Чтобы создать полезную модель, разработчики должны быть уверены, что не упустили из виду и не поняли превратно какие-то важные понятия, а для этого им нужно достаточно глубоко разбираться в предметной области. Это достигается только через непосредственное сотрудничество с людьми, знающими предметную область от и до, — то есть с бизнес-пользователями, заинтересованными лицами и экспертами в предметной области. Без этого высок риск получить техническое решение, плохо отвечающее сути задачи и непонятное специалистам бизнес-заказчика и другим разработчикам, которые будут заниматься его последующим сопровождением или расширением.

Знания собираются на маркерной доске в процессе мозговых штурмов и проработки примеров совместно с бизнес-специалистами. Это поиск, в ходе которого вырабатывается единое для всех понимание предметной области задачи, которое необходимо для создания модели, отвечающей бизнес-сценариям использования. Процесс переработки знаний, как показано на рис. 2.1, начинается с определения поведения системы. Разработчики прорабатывают варианты использования приложения с заинтересованными лицами и экспертами. Этот процесс служит катализатором общения, изучения и выработки единого понимания предмета всеми участниками. Вот почему активное участие в процессе всех заинтересованных сторон и профильных специалистов является жизненно важным.

Достижение единого понимания через общий язык

Результатом переработки знаний и артефактом единого понимания является общий единый язык. В процессе совместного моделирования с участием заинтересованных лиц и профильных специалистов каждый участник должен сознательно стремиться к систематическому использованию единого языка, охватывающего терминологию предметной области. Это должен быть четкий и ясный язык, и его необходимо использовать при описании предметной модели и предметной области задачи. Тот же язык должен фигурировать в программной реализации модели, чтобы те же термины и понятия использовались в коде в качестве имен классов, свойств и методов. Благодаря единому языку специалисты из бизнеса и разработчики смогут продуктивно общаться и понимать друг друга, обсуждая создаваемое приложение. Подробнее о едином языке рассказывается в главе 4 «Проектирование на основе модели».

ющих в разных умах и источниках, изготавливается новый интеллектуальный продукт — строгая модель, систематизирующая эти знания. У исходного термина *knowledge crunching* есть также определенный гастрономический оттенок пережевывания и переваривания, который можно заметить и в русскоязычных идиомах вроде «грызть гранит науки» или «переваривать прочитанное». — *Примеч. пер.*

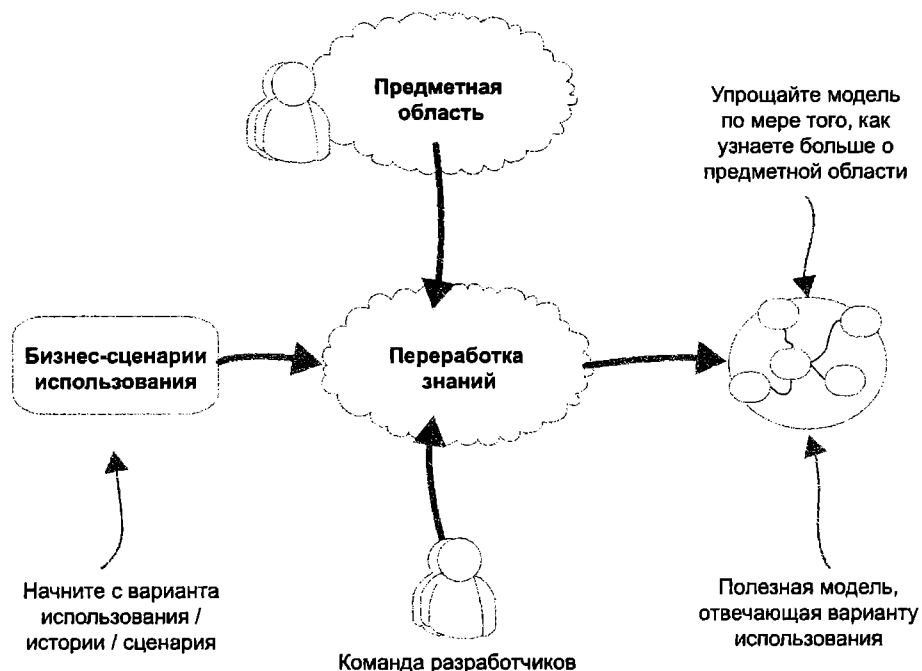


Рис. 2.1. Переработка знаний

Единый язык помогает связать программное представление модели с концептуальной моделью, которая выражена посредством текстов и диаграмм и понятна коллегам из бизнеса. Единый язык включает в себя терминологию предметной области, а также новые понятия и термины, введенные в процессе моделирования вариантов использования из предметной области. Когда все понимают, что единый язык избавляет от необходимости постоянного перевода программной модели на язык предметной модели, исчезает риск упустить что-то действительно важное.

Важность знаний в предметной области

Знания в предметной области играют даже более важную роль, чем технические ноу-хау. Команды, ведущие разработку для бизнеса со сложными процессами и логикой, должны с головой погрузиться в предметную область и как губка впитывать все знания, которые имеют отношение к сути задачи. Благодаря такому погружению они смогут сосредоточиться на том, что действительно важно, и подготовить в качестве основы для своего приложения такую модель, которая будет в полной мере отвечать бизнес-сценариям использования на протяжении всей жизни приложения.

Если вы не в состоянии простыми терминами объяснять бизнес-пользователям сложные концепции предметной области, то вы не готовы заниматься разработкой программного обеспечения для этой области. Чтобы оперативно обновлять

создаваемые вами сложные приложения, поспевая за прихотями бизнеса, постоянно меняющего свои требования, вам необходимо на этапе проектирования и разработки сфокусировать свое внимание и усилия своей команды не только на чисто технологических вопросах, но и на задачах бизнеса.

Роль бизнес-аналитиков

Может показаться, что потребность в традиционных бизнес-аналитиках в мире DDD отпала, — однако это не так. Бизнес-аналитик по-прежнему может помочь заинтересованным лицам конкретизировать их первоначальные идеи и верно описать, что должно быть на входе и выходе продукта. Кроме того, если в вашей команде есть начинающие разработчики и вы беспокоитесь о том, что будет, когда они окажутся лицом к лицу с экспертами в предметной области, вы можете попробовать привлечь бизнес-аналитиков в роли профессионалов, способных выстроить общение. Чего действительно не следует делать, так это исключать прямое общение между командой разработчиков и людьми, глубоко разбирающимися в тех или иных аспектах предметной области.

Непрекращающийся процесс

Переработка знаний — это непрекращающийся процесс; чтобы создать полезную модель, команда должна постоянно работать над упрощением картины, оставляя в центре внимания только те аспекты предметной области, которые отвечают сути задачи. Как вы узнаете в главе 4, проектирование на основе модели и развитие модели предметной области — процесс, который идет безостановочно. Вам придется отклонить множество моделей, чтобы получить модель, пригодную для текущих бизнес-сценариев использования системы. Совместная работа команды разработчиков, заинтересованных лиц и профильных специалистов из бизнеса не должна ограничиваться начальным этапом проекта. Переработка знаний с активным участием бизнеса должна продолжаться на протяжении всего времени разработки и развития приложения.

Важно сознавать также, что с каждой новой итерацией разработки системы модель будет совершенствоваться. Она будет изменяться с появлением новых требований. Новое поведение и новые варианты использования потребуют изменений в модели, а значит, разработчики и специалисты из бизнеса должны понимать, что работа над моделью никогда не закончится; это пойдет только на пользу.

С каждой итерацией команда будет все лучше понимать предметную область. За этим последуют озарения и подвижки в архитектуре, способные заметно упростить модель. После передачи системы в эксплуатацию модель должна продолжить развиваться, поскольку появятся соображения технического характера, такие как улучшение производительности, или станет лучше понятно, каким образом система используется на практике. Хорошая модель — это модель, которую легко изменить; зрелая модель сохраняет богатство и выразительность концепций и терминологии предметной области и понятна обеим сторонам — как команде разработчиков, так и специалистам из бизнеса.

Углубление знаний с помощью экспертов в предметной области

Сотрудничество разработчиков и специалистов со стороны бизнеса является важнейшим аспектом DDD и решающим условием успеха разрабатываемого продукта. Однако не менее важно найти специалистов, являющихся профильными экспертами в той области, над продуктом для которой вы работаете, и способных дать вам более глубокое понимание этой области. В терминологии DDD такие профильные специалисты называются экспертами в предметной области. Эксперты в предметной области — это люди, которые глубоко понимают свою область, от регламентов и рабочих процессов до скрытых подводных камней и разного рода тонкостей. У этих людей вряд ли будет официальное звание отраслевого эксперта; их экспертиза тесно связана с конкретной организацией. Попробуйте поискать таких людей среди владельцев продукта, пользователей и вообще всех, кто может хорошо разбираться в теме, — независимо от званий и титулов.

Эксперты в предметной области и заинтересованные лица

Пространство задачи задает набор требований, факторов на входе и ожидаемых результатов — обычно все это поступает от заинтересованных лиц. Пространство решений содержит модель, отвечающую заявленным требованиям, — и здесь вам могут принести пользу эксперты в предметной области.

Как показано на рис. 2.2, если заинтересованное лицо не является экспертом в предметной области, то его роль существенно отличается от роли эксперта. Заинтересованные лица говорят о том, что они хотят получить от системы; они смотрят прежде всего на то, что поступает в систему и что она выдает на выходе. В то же время эксперт в предметной области будет вместе с вами работать над созданием полезной модели, которая отвечает потребностям заинтересованных лиц и обеспечивает требуемое поведение приложения.

Углубление знаний специалистов со стороны бизнеса

Благодаря совместной работе не только разработчики углубляют свои знания о предметной области, в которой ведется разработка, — эксперты также оттачивают свое понимание предмета. Концепции, которые были понятны работникам организации на интуитивном уровне, получают четкие определения в руках разработчиков и экспертов в предметной области, что в конечном итоге повышает эффективность коммуникации внутри бизнеса.

Тесное взаимодействие с экспертами в предметной области

Чтобы вывести сотрудничество на новый уровень, стоит разместить команду разработчиков рядом с экспертами в предметной области. Это позволит в любой

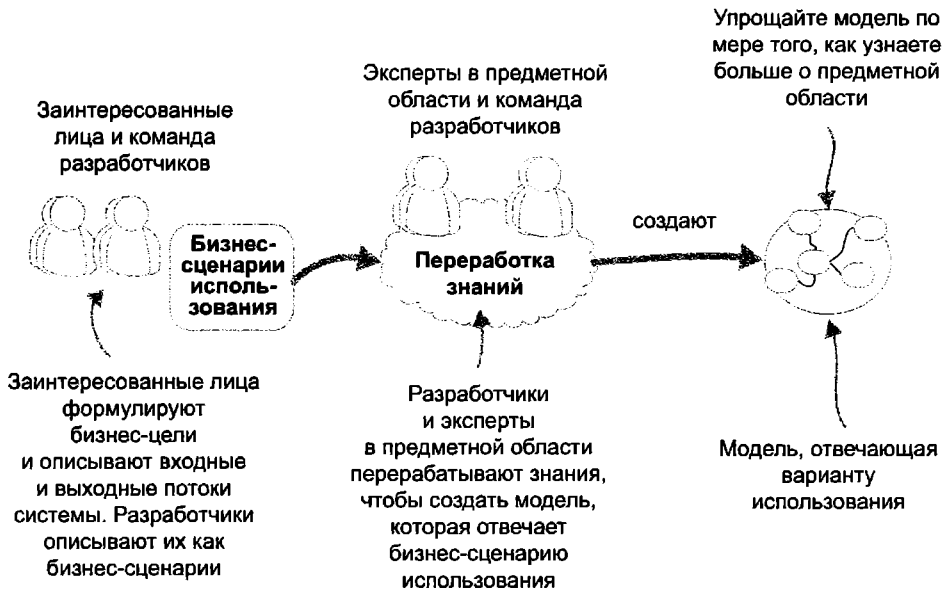


Рис. 2.2. Роли заинтересованных лиц, экспертов и разработчиков

момент оперативно получить ответы на вопросы или вовлечь экспертов в спонтанное обсуждение в коридоре либо в комнате отдыха. Когда коммуникация ограничена еженедельными проектными совещаниями, такого эффекта достичь не получится. Сотрудничество — важнейшая часть DDD; без него невозможны прорывы в строительстве архитектуры. Именно такое глубокое погружение в проектирование делает программное обеспечение полезным и гибким, позволяя ему легко адаптироваться под изменяющиеся бизнес-процессы.

Если нет возможности разместиться рядом с экспертами, присоединяйтесь к ним в обеденный перерыв. Проводите с ними столько времени, сколько это возможно, и учитесь, учитесь, учитесь. Если вы хотите преуспеть в создании высококачественного продукта, чтобы удовлетворить свое эго или добиться карьерного роста, тогда живите предметной областью, дышите ею — и вы будете щедро вознаграждены.

ПОЛЬЗУЙТЕСЬ НАЛИЧИЕМ ЭКСПЕРТА ПО МАКСИМУМУ, ПОТОМУ ЧТО ОН МОЖЕТ УЙТИ В ЛЮБОЙ МОМЕНТ

Используйте с пользой то время, когда эксперт в предметной области рядом. Не ограничивайтесь просьбами сформулировать требования или утвердить список требований, составленный вами, — активно вовлекайте своего эксперта в процесс дистилляции знаний, рисуйте с ним схемы на доске, экспериментируйте и проявляйте свое желание узнать больше о той предметной области, для которой вы создаете программное обеспечение.

Время эксперта стоит дорого. Встречая его на пороге и проявляя неподдельный интерес, вы тем самым демонстрируете ценность обмена знаниями.

Шаблоны эффективной переработки знаний

Создание удачной модели возможно только в сотрудничестве, однако бизнес-пользователи могут посчитать эту деятельность утомительной и счесть ее непроизводительной тратой времени. Бизнес-пользователи — занятые люди. Чтобы сделать встречи, посвященные переработке знаний, более увлекательными и живыми, можно использовать фасилитационные игры и другие формы сбора требований, помогающие вовлечь участников в процесс.

Сосредоточьтесь на самых интересных темах

Не вгоняйте экспертов и заинтересованных лиц в тоску, обсуждая списки требований пункт за пунктом. Как уже отмечалось, время эксперта стоит дорого. Начните с тех аспектов предметной области, которые не дают покоя коллегам со стороны бизнеса, — с тех, которые являются определяющими и для бизнеса, и для успеха вашего приложения. Например, спросите экспертов о том, какие части нынешней системы сложнее всего использовать или какие рутинные процессы не дают им заняться действительно творческой и полезной деятельностью. Какие изменения могли бы увеличить доход, повысить операционную эффективность и дать в конечном итоге экономию средств? Часто бывает полезно взглянуть на ситуацию в финансовом разрезе и посмотреть, где тратится больше всего денег или где есть препятствия к извлечению дополнительной прибыли. Обсуждение самых животрепещущих тем покажет вам, куда следует направить основные усилия по созданию единого понимания и общего языка.

Начните с вариантов использования

Изучение новой предметной области лучше начинать с создания вариантов использования. В варианте использования перечисляются шаги, необходимые для достижения цели, включая взаимодействия между пользователями и системами. Поработайте с бизнес-пользователями, чтобы понять, какие действия они выполняют в существующей системе, будь то бумажная процедура или компьютеризированный процесс. Будьте внимательны, прислушивайтесь к используемой терминологии, потому что с этого начинается формирование общего языка для описания предметной области и общения по связанным с ней вопросам. Полезно также изложить свое понимание варианта использования экспертам, чтобы они могли проверить и скорректировать ваши представления. Помните о том, что вы должны составить карту процесса в том виде, в каком он действует в реальности, разобраться в существующих потоках операций. Не пытайтесь поскорее перейти к решению, пока вы не разобрались в задаче как следует и не оценили ее в полной мере.

Задавайте продуктивные вопросы

«Как это будет выглядеть в идеальном случае? По каким критериям следует судить об успешности этого продукта? В каких случаях эти усилия будут оправданными? Чего стремится добиться организация?» Вопросы, которые вы задаете на

встречах по переработке знаний, помогут вам осознать важность продукта, который вы создаете, и разобраться в побудительных мотивах его создания.

Вот несколько примеров вопросов, которые можно задать эксперту, чтобы «разговорить» его и получить более полное представление о предметной области:

- Чем обусловлена потребность в данной системе?
- Каким образом эта система принесет пользу бизнесу?
- Что случится, если эта система не будет создана?

ПРИМЕЧАНИЕ

Грег Янг разместил в своем блоге отличную заметку о продуктивных вопросах. В комментариях к ней есть множество великолепных примеров вопросов, которые помогут вам глубже вникнуть в предметную область. Найти эту заметку можно по адресу <http://goodenoughsoftware.net/2012/02/29/powerful-questions/>.

Схемы и диаграммы

Часто люди быстрее понимают суть дела, когда перед глазами у них есть визуальное представление обсуждаемых идей. Создание набросков простых диаграмм — общепринятый прием визуализации, которым пользуются в практике DDD, чтобы повысить эффективность встреч по переработке знаний и сэкономить время экспертов в предметной области.

Начинайте с наброска на доске или на листе бумаги. Если схема будет оставаться простой и неформальной, вы сможете быстро изменять и перерисовывать ее по ходу беседы.

К сожалению, многие разработчики испытывают сложности с созданием эффективных схем и диаграмм. Есть один простой принцип, который поможет вам создавать высокоэффективные диаграммы: поддерживайте разумный уровень детализации. Обсуждая высокоуровневую концепцию — скажем, способ организовать взаимодействие независимых программных систем для выполнения бизнес-сценария, — старайтесь не уходить в концепции более низкого уровня вроде имен классов и модулей: это лишь загромодит диаграмму. Поддерживая детализацию на должном уровне, вы не перегрузите диаграмму подробностями и не упустите то, что действительно важно, а значит, участники смогут понять то, что вы пытаетесь до них донести. Часто имеет смысл создать несколько диаграмм — отдельно для каждого уровня детализации.

UML (Unified Modeling Language — унифицированный язык моделирования) — прекрасный язык, с помощью которого можно понятным образом объяснить неспециалисту устройство сложной системы. Однако для оживленных дискуссий на встречах по переработке данных его формальность может оказаться избыточной, поскольку команде предстоит многократно вносить изменения в модель. Не пытайтесь использовать для структурирования изменений модели сложные программные инструменты вроде Visio или Rational Rose, лучше набросайте модель маркером на доске. Набросок, нарисованный на ходу

за одну-две минуты, будет не так обидно переделывать, как диаграмму в Visio, на которую вы потратили все утро. Если результаты встречи по переработке знаний необходимо записать, сделайте это в конце, когда будете больше знать о предметной области.

Используйте CRC-карточки

Визуальное представление информации — действенный способ быстро разъяснять концепции и обмениваться идеями. Однако поскольку в основе философии DDD лежит идея общего языка, важно использовать приемы сбора информации, которые направлены на создание такого емкого и выразительного языка.

CRC-карточка (Class Responsibility Collaboration — «класс — ответственность — кооперация») состоит из трех частей, на которые заносится следующая информация:

- имя класса, представляющего понятие предметной области;
- область ответственности класса;
- классы, которые связаны с данным классом и нужны ему для решения его задач.

CRC-карточки направляют внимание разработчиков и специалистов со стороны бизнеса на обдумывание языка понятий предметной области.

Не спешите именовать понятия в модели

Имена играют важную роль в моделировании предметной области. Однако спешка с присвоением имен может послужить источником проблем позже, когда в процессе переработки знаний вдруг обнаружится, что то или иное понятие расходится с вашим первоначальным представлением о нем. Дело в том, что слово, выбранное в качестве имени, порождает определенные ассоциации и направляет мысли в определенное русло. Грег Янг предложил (<http://codebetter.com/gregyoung/2012/02/28/the-gibberish-game-4/>) использовать для именования тех элементов модели, в которых вы пока не уверены, выдуманные бессмысленные слова — последовательности звуков. Я предпочитаю названия цветовых оттенков, но идея здесь та же самая: вместо присвоения элементам или понятиям модели реальных имен их на время обозначают некоторыми заменителями — до тех пор, пока не станет окончательно понятно, какова их область ответственности, особенности поведения и данные, которые они представляют. Отложив именование понятий в модели на более поздний срок, вы избежите моделирования реальности, которую пытаетесь изменить ради пользы для бизнеса.

Остерегайтесь также перегрузки терминов. Избегайте имен вида XXXXService и XXXXManager. Если вы вдруг обнаружили, что постоянно добавляете в имена классов или понятий окончание Service или Manager, попробуйте подойти к именованию более творчески, постарайтесь выбирать имена, точно отражающие суть. Когда вы почувствуете, что действительно понимаете некоторую часть модели, вы сможете дать ей более точное и понятное имя.

Разработка через реализацию поведения

Разработка через реализацию поведения (Behavior-Driven Development, BDD) — это процесс разработки программного обеспечения, основанный на разработке через тестирование (Test-Driven Development, TDD). Его суть состоит в том, чтобы сконцентрироваться на описании поведения системы, а затем сформировать архитектуру извне на этой основе. Для описания поведения системы в ходе общения с экспертами в предметной области и заинтересованными лицами подход BDD использует конкретные предметные сценарии.

BDD напоминает DDD тем, что оставляет технические аспекты приложения в стороне. Разница между ними в том, что BDD концентрирует внимание разработчика на поведении программного обеспечения, на том, как система должна вести себя, а DDD помещает в центр внимания модель предметной области, лежащую в основе программного обеспечения и определяющую его поведение. Различие тонкое, но важное.

В BDD имеется свой единый язык для определения требований (аналитический язык, если угодно), известный как GWT (Given, When, Then — «дано, если, тогда»). Формат GWT помогает структурировать общение с экспертами в предметной области и выявлять реальное поведение предметной области.

Чтобы увидеть BDD в деле, посмотрите, как можно выявить требования к продукту на основании пользовательских историй.

Вот пример описания одной функции сайта электронной коммерции:

Функция: бесплатная доставка для крупных заказов

Одна из историй, описывающих эту функцию, могла бы быть изложена так:

*Чтобы поднять среднюю сумму заказа, составляющую сейчас \$50,
как руководитель службы сбыта*

я хотел бы предложить клиентам бесплатную доставку для заказов стоимостью \$60 и больше.

Еще один пример истории:

Чтобы использовать особенности типичного поведения покупателей в разных странах,

как руководитель службы сбыта

я хотел бы определять порог суммы заказа с бесплатной доставкой для каждой страны доставки в отдельности.

Здесь описывается поведение, имеющее значение для коммерции. В описание включены также роль и выгода. Указание роли, связанной с данной функцией, позволяет разработчикам и экспертам в предметной области понять, к кому следует обратиться и кому передать полномочия. Указание выгод обосновывает потребность в наличии такой функции и помогает разобраться, зачем она нужна бизнес-пользователю.

Чтобы лучше понять ту или иную функцию и ее поведение, используются сценарии BDD, описывающие эту функцию в контексте различных вариантов использования. Сценарии начинаются с характеристики исходной ситуации — условия

«Дано» (Givens). Далее в сценарии описываются одно или более событий «Если» (Whens), а затем — ожидаемые результаты «Тогда» (Thens).

Вот пример такого сценария BDD:

Сценарий: сумма заказа клиента превосходит порог, дающий право на бесплатную доставку

Дано: минимальная сумма заказа, дающая право на бесплатную доставку, равна \$60

И: я клиент, в корзине которого в настоящий момент лежат товары на сумму \$50

Если: я добавлю в корзину товар стоимостью \$11

Тогда: мне должна быть предложена бесплатная доставка

Другой пример:

Сценарий: сумма заказа клиента ниже порога, дающего право на бесплатную доставку, но клиент совершил действие, которое вызывает предложение приобрести еще что-нибудь

Дано: минимальная сумма заказа, дающая право на бесплатную доставку, равна \$60

И: я клиент, в корзине которого в настоящий момент лежат товары на сумму \$50

Если: я добавил в корзину товар стоимостью \$9

Тогда: меня должны известить, что при увеличении заказа еще на \$1,00 мне будет предложена бесплатная доставка

Такие сценарии не только представляют собой простой способ зафиксировать обнаруженные требования, но и задают критерии, с помощью которых разработчики и тестировщики смогут оценить полноту реализации данной функции, а бизнес-пользователи — проверить, верно ли команда поняла ее суть.

Такой метод описания требований устраняет неоднозначности, иногда возникающие при традиционном документировании требований, а также выводит на первый план язык предметной области. Описание функций и сценарии являются продуктом сотрудничества команды разработчиков и специалистов со стороны бизнеса и помогают формировать единый язык.

Быстрое прототипирование

Используйте в ходе встреч по переработке знаний быстрое прототипирование. Мало что так нравится бизнес-пользователям, как макеты экранов: они гораздо полнее всего остального раскрывают те намерения, которые послужили поводом для создания продукта. Пользовательский интерфейс понятен пользователям, они могут взаимодействовать с ним и наглядно разыгрывать разные рабочие ситуации.

Другая форма быстрого прототипирования — реализация требований в коде. Грег Янг называет такой код аналитическим; его презентацию, посвященную этому вопросу, можно найти по адресу <http://skillsmatter.com/podcast/open-source-dot-net/>

mystery-ddd. Бизнес-пользователям нравится, когда что-то осязаемое создается на их глазах. Начав писать код, вы сможете сделать обсуждения более предметными. Реализуя на практике абстрактные идеи, полученных в ходе переработки знаний, вы сможете проверить и подтвердить правильность своей модели. Это поможет также предотвратить уход в абстрагирование, чреватый параличом анализа (analysis paralysis) (<http://sourcemaking.com/antipatterns/analysis-paralysis>).

Написание кода по ходу дела помогает формулировать продуктивные вопросы и обнаруживать пропущенные варианты использования. Используйте этот код для выявления и решения проблем. Примерно через час после начала обсуждения посмотрите, нельзя ли создать программную модель по результатам обсуждения. По моему опыту, реализация идей в программном коде помогает зафиксировать понятия предметной области и подтвердить правильность модели. Кроме того, этот процесс помогает поддерживать вовлеченность разработчиков и увлечь их освоением предметной области, поскольку они практически немедленно начинают получать реакцию на предложенную архитектуру.

Запомните: программная модель должна создаваться только в рамках конкретного контекста для решения конкретной задачи; построить эффективную модель целой предметной области невозможно. Думайте локально, отталкивайтесь от правил и затем постепенно расширяйте модель. И самое главное — не забывайте о том, что вы пишете код, который позже отправится в мусорную корзину. Не останавливайтесь на первой работоспособной модели и не позволяйте себе привязаться к первой же хорошей идее.

Изучайте «бумажные» процессы

Разрабатывая решение для предметной области, где пока еще нет программного решения, наблюдайте, как специалисты используют язык в текущем «бумажном» рабочем процессе. Усложнение модели для обработки крайних случаев может пойти во вред некоторым процессам и рабочим процедурам. Возможно, редкие крайние ситуации лучше обрабатывать вручную, по старинке; включая их в модель, вы можете потратить много сил с сомнительной пользой для бизнеса.

Ищите уже созданные модели

Иногда можно не изобретать велосипед. Если предметная область, с которой вы работаете, существует достаточно давно (например, если это банковская деятельность), для нее наверняка уже есть известные модели. У вас нет времени на то, чтобы становиться экспертом в предметной области, поэтому ищите информацию, которая поможет вам больше узнать о ней. В своей книге «Analysis Patterns: Reusable Object Models» (Addison-Wesley, 1996) Мартин Фаулер предлагает множество типовых моделей для различных предметных областей, которые можно использовать как отправную точку в обсуждениях.

Модели предметной области могут уже существовать в самой организации. Узнайте, нет ли готовой карты процессов и диаграммы потоков работ, — эти документы

помогут вам глубже понять предметную область. Сформируйте вики-подобную базу знаний с терминами и определениями, используемыми в организации, и поделитесь ею с командой. Помните, что уровень команды определяется уровнем самого слабого разработчика в ней; это касается не только технических навыков, но и знаний в предметной области.

ПЫТАЙТЕСЬ СНОВА И СНОВА

Вы вряд ли получите хорошую модель с первой же попытки; неудачной может оказаться и вторая попытка, и третья. Не бойтесь экспериментировать. Привыкайте выбрасывать неудачные проекты и начинать все заново. Помните, что нет единственной истинно правильной модели — есть лишь модель, полезная для решения данного круга задач в данном контексте.

Разберитесь в истинных намерениях

Когда клиенты требуют внести улучшения в существующее программное обеспечение, это повод насторожиться: их требования нередко продиктованы ограничениями существующих систем, а не тем, что им действительно нужно. Спросите себя, как часто вы обращались к пользователям, чтобы разобраться в мотивах, стоящих за их требованиями. Было ли вам ясно, *почему* они хотят то, что хотят? Поняв истинные потребности клиента и сформулировав их, вы нередко сможете предложить более удачное решение. Когда это происходит, клиенты обычно удивляются и произносят сакраментальное «Ой, правда? А я и не думал, что такое возможно!» Помните: вы — движущая сила проекта. Не нужно слепо идти за требованиями пользователей. Бизнес-пользователи не всегда способны внятно и доходчиво описать функции или цели. Чтобы продукт обладал действительной ценностью для бизнеса, вы должны понимать и разделять ту мысленную картину, которая стоит за словами пользователей, и принимать во внимание цели бизнеса.

Событийный штурм

Событийный штурм (Event Storming) — это веселая и увлекательная процедура, помогающая команде разработчиков и бизнес-специалистов быстро проникнуть в предметную область. Она представляет собой групповую совместную работу, в ходе которой разработчики и эксперты в предметной области формируют единое видение предметной области, причем разработчики задают вопросы, а от экспертов поступают ответы. Для переработки знаний нужно открытое помещение с просторным полем для визуального моделирования — это могут быть несколько маркерных досок или большой рулон бумаги (вполне подойдет оберточная).

Исследование предметной области начинается с некоторого предметного события — события, возникающего в предметной области и существенного для бизнеса. Стикер, представляющий это предметное событие, приклеивается на поле для моделирования (доску или бумагу), после чего внимание участников пере-

ключается на триггеры этого события. Событие может быть вызвано каким-либо действием пользователя — такое действие заносится на поле как «команда». Источником или причиной события могут служить внешняя система или другое событие — их также добавляют на поле. Работа продолжается до тех пор, пока не будут исчерпаны все вопросы. Затем команда может приступать к построению модели, отталкиваясь от точек принятия решений по поводу событий, которые в свою очередь сами порождают другие события.

Событийный штурм — чрезвычайно эффективная методика выработки единого языка, поскольку каждое событие и каждая команда должны по ходу дела явным образом получить название, что способствует взаимопониманию между разработчиками и бизнес-специалистами. Эта методика может также выявить подобласти и смысловое ядро в предметной области, о чем подробно рассказывается в главе 3 «Концентрация на смысловом ядре». Ключевое ее преимущество, однако, состоит в том, что это увлекательное действо, которое затягивает участников и легко организуется. Создал методику событийного штурма Альберто Брандолини (Alberto Brandolini), и за дополнительной информацией о ней вы можете обратиться к материалам его блога, расположенного по адресу <http://ziobrando.blogspot.co.uk/>.

Составление карты воздействий

Составление карты воздействий (impact mapping) — новая методика, помогающая лучше понять намерения заинтересованных лиц. При составлении карты воздействий вы выходите за рамки традиционного описания требований и пытаетесь выяснить, на что и каким образом бизнес пытается повлиять. Целью бизнеса является увеличение продаж? Или расширение своей доли на рынке? Или выход на новый рынок? А может быть, компания хочет увеличить вовлеченность клиентов, чтобы получить больше лояльных клиентов с высокой пожизненной ценностью¹?

Уяснив, какое воздействие пытается осуществить бизнес, вы сможете более эффективно помочь ему в этом. С точки зрения DDD здесь важно то, что вы, зная цели бизнеса, сможете задавать более правильные вопросы в процессе переработки знаний.

Как ни странно, составление карты воздействий — довольно неформальная методика. Вы просто структурируете ключевую информацию о целях бизнеса и путях их достижения с помощью диаграмм, напоминающих ассоциативные карты. Эта работа выполняется совместно со специалистами со стороны бизнеса, а поэтому, как и переработка знаний, способствует формированию единого представления о продукте. На рис. 2.3 показан пример карты воздействий, которая представляет желание компании, занимающейся электронной коммерцией, увеличить объем продаж велосипедов на 25%.

Карта воздействий, как можно увидеть, начинается с воздействия. На рис. 2.3 в этом качестве выступает увеличение продаж велосипедов на 25%. С воздействием

¹ Пожизненная ценность клиента (customer lifetime value, CLV) — важный маркетинговый показатель, представляющий собой оценку вероятной будущей чистой прибыли, которую компания ожидает получить от конкретного потребителя за все то время, пока он остается клиентом компании. — *Примеч. ред.*

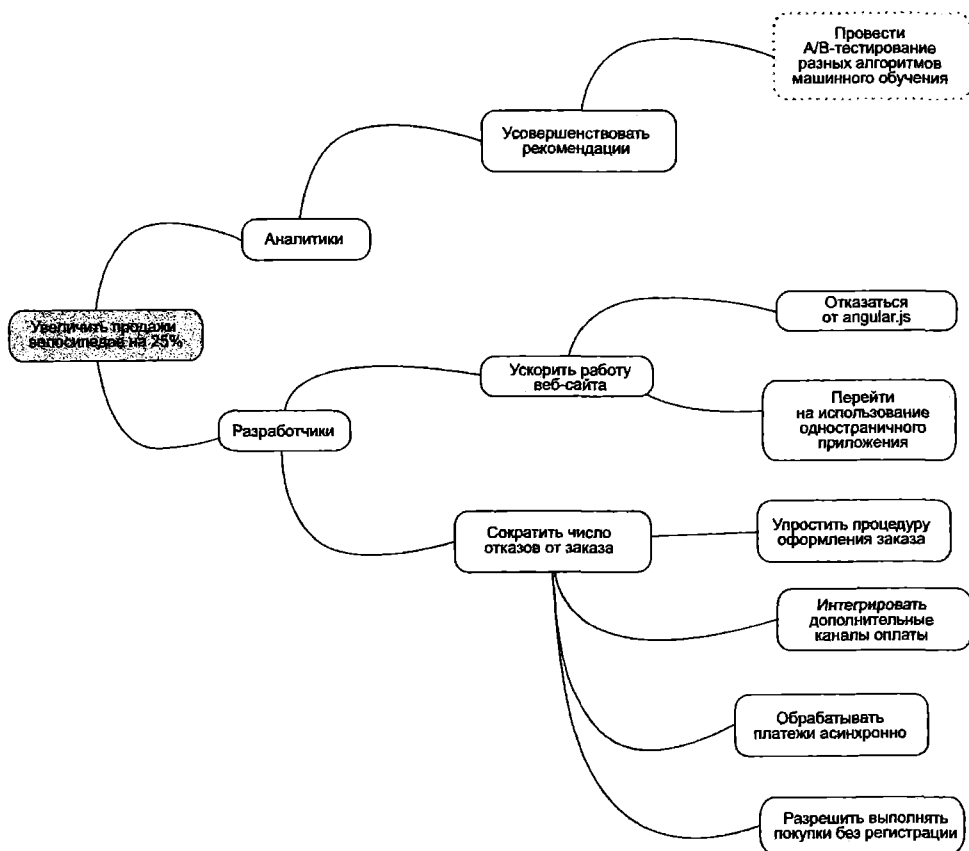


Рис. 2.3. Карта воздействий

непосредственно связаны действующие лица — люди, которые могут внести какой-либо вклад в это воздействие. На рис. 2.3 такими людьми являются аналитики и разработчики. Узлы, следующие за действующими лицами, описывают, чем именно могут поспособствовать воздействию эти действующие лица. Например, разработчики могут содействовать росту продаж, ускорив работу веб-сайта и упростив его, чтобы покупатели охотнее им пользовались и реже отказывались от заказа на полпути. Наконец, на последнем уровне иерархии перечислены конкретные задачи, которые нужно решить. Как показано на рис. 2.3, один из путей ускорить работу сайта — отказаться от использования медленных фреймворков.

Во многих проектах разработчики видят только нижние ярусы карты воздействий — что именно бизнес хочет от разработчиков и как это, по мнению бизнеса, должно быть сделано. Имея перед глазами карту воздействий, можно понять ход мысли руководителей организации и увидеть, чего они пытаются добиться на самом деле, а затем, опираясь на собственный опыт и технические познания, предложить более удачные альтернативы, которые, возможно, никогда не пришли бы заказчику в голову.

Некоторые приверженцы DDD очень высоко оценивают методику составления карты воздействий вне зависимости от того, применяется она совместно с DDD или отдельно. Мы настоятельно рекомендуем познакомиться с этой методикой подробнее, приобретя книгу Гойко Аджича (Gojko Adzic) «Impact Mapping» или посетив веб-сайт (<http://www.impactmapping.org/>).

Бизнес-модель организации

Бизнес-модель содержит массу полезной информации о предметной области и подчеркивает основные цели бизнеса. К сожалению, мало кто из разработчиков тратит свое время на то, чтобы ознакомиться с бизнес-моделью своего заказчика или хотя бы просто понять, что такое бизнес-модель в принципе.

Один из лучших способов разобраться в бизнес-модели организации — изобразить ее с помощью шаблона бизнес-модели (Business Model Canvas). Этот чрезвычайно действенный метод визуализации предложили Александр Остервальдер (Alexander Osterwalder) и Ив Пинье (Yves Pigneur) в своей авторитетной книге «Построение бизнес-моделей»¹; книга написана доступным языком, и мы настоятельно рекомендуем ее разработчикам.

Шаблон бизнес-модели делит бизнес-модель организации на девять блоков, как показано в таблице (см. с. 64), где изображена бизнес-модель компании, торгующей спортивным снаряжением.

Изучив эти девять блоков, вы поймете, что важно для бизнеса. Здесь содержится ключевая информация, в частности сведения о том, из чего складываются доходы организации, каковы ее самые важные активы и, что самое главное, кто ее клиенты. Каждый из блоков модели коротко описан ниже. За более подробной информацией обращайтесь к книге «Построение бизнес-моделей».

- Потребительские сегменты (Customer Segments) — разные виды клиентов, представляющих интерес для бизнеса, например: рыночные ниши, массовые рынки, корпоративные клиенты.
- Ценностные предложения (Value Propositions) — продукты или услуги, которые компания предлагает своим клиентам, например: материальные товары, облачный хостинг.
- Каналы сбыта (Channels) — как бизнес доставляет клиентам товары или услуги, например: физическая доставка товаров, веб-сайт.
- Взаимоотношения с клиентами (Customer Relationships) — виды взаимоотношений, которые есть у бизнеса с каждым потребительским сегментом, например: персональная помощь, автоматизированная электронная служба поддержки.
- Потоки поступления доходов (Revenue Streams) — различные пути получения доходов, например: доходы от рекламы, абонентские платежи.
- Ключевые ресурсы (Key Resources) — наиболее ценные активы организации, например: интеллектуальная собственность, ключевые сотрудники.

¹ Остервальдер А., Пинье И. Построение бизнес-моделей: Настольная книга стратега и новатора. — М.: Альпина Паблишер, 2012. — *Примеч. пер.*

Ключевые партнеры <ul style="list-style-type: none">Известный спортсмен АИзвестный спортсмен БПоставщик спортивного снаряжения АПоставщик спортивного снаряжения БКомпания-перевозчикПлатежная система	Ключевые виды деятельности <ul style="list-style-type: none">МаркетингВыработка рекомендаций	Ценностные предложения <ul style="list-style-type: none">Профессиональное спортивное снаряжениеЛюбительское спортивное снаряжение	Взаимоотношения с клиентами <ul style="list-style-type: none">Персональная помощьСлужба поддержки, горячая телефонная линия	Потребительские сегменты <ul style="list-style-type: none">Профессиональные спортсменыЭнтузиасты, любители фитнесаШирокие массы
	Ключевые ресурсы <ul style="list-style-type: none">БрендОбширный каталогПрограмма лояльности		Каналы сбыта <ul style="list-style-type: none">Веб-сайт	
Структура издержек <ul style="list-style-type: none">Товарные запасыЗарплата сотрудниковСклады/имуществоСпонсирование спортсменов / маркетинг			Потоки поступления доходов <ul style="list-style-type: none">Продажа товаровРеклама	

- Ключевые виды деятельности (Key Activities) — виды деятельности, лежащие в основе функционирования бизнеса, например: разработка программного обеспечения, анализ данных.
- Ключевые партнеры (Key Partnerships) — список наиболее ценных партнеров предприятия, например: поставщики, консультанты.
- Структура издержек (Cost Structure) — расходы, которые несет компания, например: заработная плата сотрудников, подписка на программное обеспечение, хранение товарных запасов.

Вооружившись информацией, представленной в шаблоне бизнес-модели, вы сможете задавать экспертам в предметной области осмысленные вопросы и тем самым содействовать развитию компании за пределами чисто технических аспектов. Понимание бизнес-модели компании стоит тех небольших усилий, которые придется потратить, чтобы найти эту модель и разобраться в ней.

Целенаправленное открытие

Дэн Норт (Dan North), создатель методики BDD, опубликовал метод, помогающий увеличить объем и повысить качество знаний о предметной области. Этот метод называется «Целенаправленное открытие» (Deliberate Discovery) (<http://dannorth.net/2010/08/30/introducing-deliberate-discovery/>). На стадиях планирования и сбора требований вам нужно не заикливаться на методиках гибкой раз-

работки вроде покера планирования и сбора пользовательских историй¹, а потратить время на изучение тех аспектов предметной области, с которыми вы плохо знакомы. Дэн считает, что «невежество — единственное серьезное препятствие на пути к успеху». А значит, чем больше вы знаете о предметной области, тем эффективнее будет ваша работа над моделью.

В начале проекта разработчикам необходимо приложить согласованные усилия для того, чтобы выявить те аспекты предметной области, о которых им ничего или почти ничего не известно, — ими необходимо вплотную заняться в процессе переработки знаний. Во время встреч по переработке знаний команды продолжают поиски неизвестного, выявляя новые аспекты предметной области, не обнаруженные прежде. Направлять этот процесс должны эксперты в предметной области и заинтересованные лица, которые помогут разработчикам сосредоточиться на наиболее важных аспектах, а не перелопачивать всю предметную область. Это позволит команде выявить пробелы в знаниях предметной области и быстро заполнить их.

Водоворот исследования модели

Эрик Эванс, создатель философии предметно-ориентированного проектирования, написал черновой вариант документа под названием «Model Exploration Whirlpool» (водоворот исследования модели: <http://domainlanguage.com/ddd/whirlpool/>). В документе описывается метод моделирования и переработки знаний, который дополняет другие методики гибкой разработки и может по мере необходимости применяться на любом этапе разработки приложения. Этот метод используется не для моделирования как такового, а скорее для решения проблем, возникающих при создании модели. Такие явления, как проблемы в коммуникации с бизнесом, чрезмерно сложная архитектура решения или острая нехватка знаний о предметной области, служат красноречивыми признаками того, что следует приступить к процессу, определение которого дано в документе «Model Exploration Whirlpool», и провести еще один цикл переработки знаний предметной области.

В «водоворот» включаются следующие действия:

○ Исследование сценария.

Эксперт в предметной области описывает сценарий, который беспокоит разработчиков или вызывает у них сложности с пониманием предметной области. Сценарий — это последовательность шагов или процессов, важная для эксперта, ключевая для приложения и попадающая в область охвата проекта. После того как эксперт объясняет сценарий на конкретных примерах, понятных команде, разработчики представляют его в виде визуальных диаграмм, как при событийном штурме.

○ Моделирование.

Одновременно с прохождением сценария разработчики начинают исследовать текущую модель и оценивают, насколько она полезна для выполнения сценария, описанного экспертом.

¹ См. https://ru.wikipedia.org/wiki/Покер_планирования, https://ru.wikipedia.org/wiki/Пользовательские_истории. — Примеч. пер.

○ Проверка модели.

После того как разработчики пересмотрели модель или создали новую, они исследуют ее применимость на других сценариях, предложенных экспертом, чтобы проверить ее пригодность.

○ Подведение итогов и документирование.

Важные сценарии, помогающие продемонстрировать работу модели, должны быть зафиксированы в документации. Из ключевых сценариев формируется набор эталонных сценариев для демонстрации того, как модель справляется с решением ключевых задач в предметной области. Бизнес-сценарии изменяются намного реже, чем модель, поэтому полезно составить набор наиболее важных сценариев, которые будут использоваться в качестве эталонных для проверки модели при любом ее изменении. Однако не нужно пытаться задокументировать каждое проектное решение и каждую модель; некоторые идеи должны остаться на маркерной доске.

○ Создание пробного кода.

Достигнув более глубокого понимания предметной области и совершив очередной прорыв в проектировании, разработчики должны попытаться написать программный код, доказывающий возможность реализации новых идей.

Ключевые идеи

- Переработка знаний — это искусство обработки информации о предметной области с целью выявления тех ее аспектов, которые связаны с сутью задачи и могут быть использованы для построения практической модели.
- Знания приобретаются разработчиками в сотрудничестве с экспертами в предметной области. Сотрудничество помогает заполнить пробелы в знаниях и сформировать единое видение предметной области.
- Единое понимание предметной области достигается через общий язык, известный как единый язык.
- Переработка знаний — непрекращающийся процесс; сотрудничество и общение со специалистами организации не должны ограничиваться начальным этапом проекта. Глубокое проникновение в суть и прорывы в проектировании происходят только после многократной совместной проработки задачи.
- Знания приобретаются всеми членами команды на любом этапе проектирования — у маркерной доски, у бачка с питьевой водой, во время мозговых штурмов и в ходе совместной работы над прототипами.
- Эксперты в предметной области — это специалисты организации. К их числу относится любой, кто может дать ценные сведения о предметной области (пользователи, владельцы продукта, бизнес-аналитики, внутренние разработчики и технические специалисты).
- Заинтересованные лица формулируют требования к приложению, но едва ли могут подробно ответить на все вопросы, касающиеся предметной области.

Поэтому, приступая к проектированию моделей для центральных или сложных аспектов предметной области, обращайтесь к экспертам в предметной области.

- Привлекайте экспертов в предметной области к проработке наиболее важных частей системы. Не ограничивайтесь простым зачитыванием требований из списка с просьбой прокомментировать их.
- Закладывайте изменение модели в план; не привязывайтесь слишком сильно к одной какой-то модели, потому что новые сведения, полученные в процессе переработки знаний, могут сделать эту модель устаревшей.
- Работая с экспертами в предметной области, сосредоточьтесь на самых важных аспектах предметной области; уделяйте максимум внимания тем аспектам, которые обеспечат успех приложения.
- Управляйте процессом переработки знаний так, чтобы охватить наиболее важные варианты использования системы. Просите экспертов в предметной области пройти по конкретным сценариям для вариантов использования, чтобы восполнить недостающие знания.
- Формулируйте продуктивные вопросы и разбирайтесь в намерениях бизнеса. Нужно не просто реализовывать набор требований, а активно взаимодействовать с бизнесом; работайте *вместе* с ним, а не только *для* него.
- Визуализируйте полученные знания с помощью схем и приемов событийного штурма. Визуализация предметной области поможет сделать сотрудничество с бизнес-специалистами более тесным и превратить переработку знаний в увлекательный процесс.
- Используйте приемы разработки через реализацию поведения (BDD), чтобы сосредоточить свое внимание на поведении приложения, а внимание экспертов в предметной области и заинтересованных лиц — на конкретных сценариях. Методика BDD является отличным катализатором общения с экспертами и заинтересованными лицами. В ней есть стандартные языковые конструкции, помогающие выделить поведение и дать ему практическое описание.
- Пишите экспериментальный код, чтобы проверить работоспособность модели и оценить последствия компромиссных решений, принятых из технических соображений.
- Ищите в наработках отрасли уже готовые процессы и модели, чтобы не изобретать велосипед и быстрее получить знания о предметной области.
- Выясняйте, чего вы еще не знаете, заблаговременно выявляйте пробелы в знаниях команды и планомерно заполняйте их с помощью методики целенаправленного открытия. Как можно раньше ликвидируйте нехватку знаний в предметной области.
- Если вам необходимы рекомендации по исследованию модели, воспользуйтесь водоворотом исследования модели, опираясь на материалы статьи «Model Exploration Whirlpool» Эрика Эванса (<http://domainlanguage.com/ddd/whirlpool/>). Действия, предлагаемые этой методикой, особенно полезны, когда возникают проблемы в коммуникации с бизнесом, архитектура решения чрезмерно усложняется или команда отчетливо ощущает нехватку знаний о предметной области.

3

Концентрация на смысловом ядре

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Зачем нужна дистилляция объемной предметной области
- Как выявить смысловое ядро
- Как сконцентрировать усилия на смысловом ядре
- За что отвечают неспециализированные и поддерживающие области
- Почему не все части системы требуют детального проектирования

Важно понимать, что не все аспекты задачи равноценны: одни части приложения важнее других. Чтобы приложение стало успешным, на некоторые его части нужно обратить более пристальное внимание и потратить больше сил. В процессе переработки знаний с экспертами в предметной области важно отсечь все несущественное, чтобы сконцентрироваться на самом важном. Методика проектирования на основе модели сложна, и ее следует использовать только для реализации тех частей системы, которые критически важны для ее успеха. Эта глава рассказывает, как выявить наиболее важные части системы и как сосредоточиться на них, используя прием дистилляции. Зная, какие части требуют повышенного внимания, вы сможете выполнить более глубокое моделирование того, что действительно существенно, и направить силы на самое важное.

Зачем нужна декомпозиция предметной области

Крупные системы, построенные для сложных предметных областей, часто представляют собой сочетание компонентов и подсистем, каждая из которых совершенно необходима для того, чтобы работала вся система. Однако некоторые части решения более ценны, чем остальные, а потому важно уметь сосредоточить силы и внимание на тех аспектах, которые наиболее важны для бизнеса. Невозможно

в равной степени уделить внимание всем частям системы, добиваясь высокого качества, — но это и не нужно. Попытки равномерной проработки системы приведут к тому, что действительно важные области не получают должного внимания.

Чтобы определить самые значимые разделы предметной области, нужно провести ее дистилляцию и выделить смысловое ядро. Разделив большую предметную область на подобласти, можно более эффективно распределить ресурсы между разными частями, позаботившись о том, чтобы самые талантливые и умелые разработчики занимались прежде всего теми частями системы, которые наиболее важны для бизнеса, а не теми, которые сложны технически или опираются на новые фреймворки либо малознакомую инфраструктуру. Кроме того, подобласти, выделенные из большой предметной области, определяют пути и способы проектирования решения.

ДОЛЖНА ЛИ ВСЕМ УПРАВЛЯТЬ ОДНА МОДЕЛЬ?

Может показаться разумным построить единую модель для всей предметной области. Однако это может оказаться очень сложной задачей, потому что такая модель должна отвечать всем потребностям предметной области. Из-за этого единая модель получается либо слишком сложной, либо слишком общей и лишенной какого-либо поведения. При разработке больших систем лучше разделить большое пространство задачи на более мелкие специализированные модели, которые можно связать с конкретным контекстом, — это повысит управляемость модели. Не забывайте, что суть DDD заключается в снижении сложности. Единая монолитная модель может увеличить сложность, поэтому предметную область следует разделить так, чтобы в пространстве решения можно было построить более мелкие модели.

Как выделить суть задачи

Чтобы знать, куда направить свои усилия, нужно уяснить, ради чего вообще создается приложение. Вам необходимо понять, в чем состоит бизнес-стратегия и каким образом программное обеспечение, которое вы создаете, будет способствовать ее реализации. Почему было принято решение создать заказной программный продукт, а не приобрести готовое коммерческое приложение? Каким образом создание этого приложения поможет бизнесу? Как оно соотносится со стратегией компании? Почему разработка приложения ведется внутри, а не заказывается на стороне? Даст ли бизнесу какие-либо конкурентные преимущества часть этого программного обеспечения?

Старайтесь понять, что стоит за требованиями

Когда клиенты требуют внести улучшения в существующее программное обеспечение, это повод насторожиться: их требования нередко продиктованы

ограничениями существующих систем, а не тем, что им действительно нужно. Спросите себя, как часто вы обращались к пользователям, чтобы разобраться в мотивах, стоящих за их требованиями.

Составьте обзор предметной области, в котором обозначены ключевые моменты

Прежде чем приступить к работе над продуктом, обязательно попросите предоставить обзорную документацию проекта. В любой крупной организации проект начинается задолго до привлечения разработчиков и часто есть короткий документ-обоснование — обзор проекта, описывающий, почему компания отдала предпочтение именно этому направлению разработки. В этом обосновании нередко можно найти ключ к определению смыслового ядра. В нем объясняется, почему создание этого приложения — хорошая идея; изучите его и выделите самые важные моменты. Запишите эти моменты на доске, чтобы все члены команды видели и понимали, для чего они пишут программное обеспечение.

На начальном этапе проекта можно составить концептуальный обзор предметной области (domain vision statement), чтобы явным образом обозначить, что имеет решающее значение для успеха программного продукта, какова цель бизнеса и в чем заключается ценность продукта. Следует ознакомить с этим изложением разработчиков и, возможно, даже поместить его на стену рабочего помещения в виде плаката — как напоминание о том, для чего создается программное обеспечение.

ПОДХОД К РАЗРАБОТКЕ ПРОДУКТОВ В КОМПАНИИ AMAZON

В Amazon используется уникальный подход к созданию обзора предметной области, получивший название «*работа вспять*» (*working backwards*; см. <http://www.quora.com/What-is-Amazons-approach-to-product-development-and-product-management>). Перед разработкой новых расширений менеджер по продукту выпускает внутренний пресс-релиз, анонсирующий законченный продукт, где перечислены новые функции и преимущества, которые они дают. Если клиенту эти преимущества не кажутся привлекательными или достойными внимания, менеджер по продукту переносит пресс-релиз до тех пор, пока они не станут для клиента по-настоящему ценными. Тем самым компания Amazon постоянно удерживает клиента в центре внимания и еще до начала разработки ясно представляет, чем полезна та или иная новая функция.

Как сосредоточиться на главной задаче

Большую предметную область можно разделить на несколько подобластей, чтобы уменьшить сложность и отделить наиболее важные части от остальной системы.

На рис. 3.1 показана схема разделки свиной туши из предметной области мясной торговли, напоминающая разделение пространства задачи на части. Понимание подобластей системы поможет вам разделить пространство задачи. Подобласти — это абстрактные понятия; не следует путать их с организационной структурой компании. Подобласти очерчивают сферы компетенции, задают бизнес-процессы и служат представлением функциональности системы.

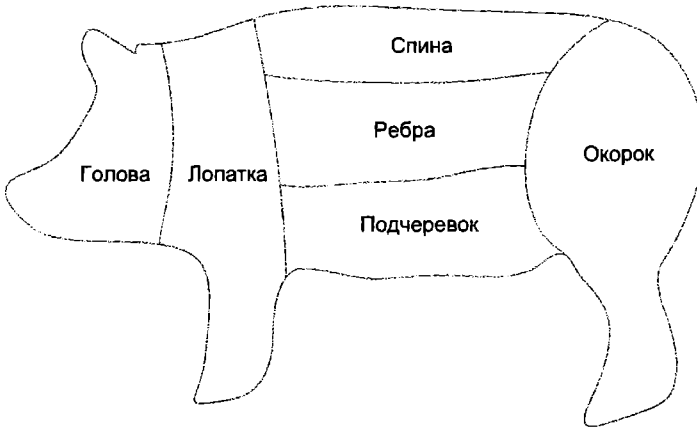


Рис. 3.1. Схема разделки свиной туши

Постарайтесь при обсуждении пространства задачи в ходе его анализа отложить в сторону технические соображения. Соображения безопасности — это технические соображения, если только сама безопасность не является пространством задачи. То же касается журналирования и маршрутов аудита — это инфраструктурные тонкости. Сосредоточьтесь прежде всего на предметной области.

Путем дистилляции предметной области вы уменьшаете ее сложность, разделяя задачу и тем самым обретая власть над ней. Модели меньшего размера могут быть созданы и осмыслены в контексте подобластей. Это избавляет от необходимости определять единую большую модель, представляющую всю предметную область задачи целиком. Многие из полученных подобластей, например составление отчетов или уведомления, могут оказаться универсальными, подходящими для любого корпоративного программного обеспечения. Такие подобласти, которые не являются определяющими для данного приложения, называют неспециализированными предметными областями (*generic domains*). Те части предметной области, которые отличают вашу продукцию от продукции конкурентов и составляют ее конкурентные преимущества, являются смысловым ядром (*core domains*). Именно смысловое ядро служит основанием для создания собственного программного обеспечения. Остальные подобласти, входящие в состав крупномасштабных приложений, называют поддерживающими предметными областями (*supporting domains*) — они обеспечивают нормальное функционирование смыслового ядра и всей системы в целом.

Дистилляция предметной области задачи

Возьмем для примера предметную модель сайта электронных торгов, изображенную на рис. 3.2. Эта система в целом состоит из множества компонентов. Некоторые из них можно найти в любой онлайн-системе, но некоторые уникальны для данной предметной области и конкретного бизнеса.



Рис. 3.2. Предметная область сайта электронных торгов

На рис. 3.3 показано, как большая предметная область делится на подобласти. Раздел «Членство» представляет ту часть системы, которая отвечает за регистрацию новых членов (пользователей), сохранение информации о них, их настроек и предпочтений. Раздел «Продавец» охватывает все процессы и функции, имеющие отношение к деятельности продавца. Раздел «Аукцион» — та часть предметной области задачи, которая отвечает за хронометраж аукционов и процесс торгов. Раздел «Списки» — это каталоги лотов, доступных на сайте электронных торгов. Наконец, подобласть «Урегулирование споров» связана с разногласиями между членами и продавцами.

Задача дистилляции знаний (distillation of knowledge) после рабочих встреч с экспертами в предметной области — помочь выявить то важное и уникальное, что должно быть в создаваемом приложении. Как показано на рис. 3.4, каждую из полученных подобластей можно охарактеризовать как основную (смысловое ядро), неспециализированную или поддерживающую.

На рис. 3.4 видно, что смысловым ядром сайта электронных торгов являются подобласти «Продавец» и «Аукцион». В подобласти «Продавец» находятся рейтинги продавца и предметная логика определения отчислений, вносимых продавцом. Подобласть «Аукцион» обеспечивает механизм проведения аукциона и обработки заявок покупателей. Оба этих раздела жизненно важны для успеха сайта электронных торгов. Разделы «Членство» и «Списки» обеспечивают поддержку смыслового ядра, предоставляя покупателям возможность создавать учетные



Рис. 3.3. Предметная область сайта электронных торгов, дистиллированная в подобласти



Рис. 3.4. Предметная область сайта электронных торгов, дистиллированная в подобласти — основные (смысловое ядро), неспециализированные и поддерживающие

записи и находить нужные им товары. Раздел «Урегулирование споров» является неспециализированным и может быть реализован с использованием готового пакета стороннего производителя; в данном случае речь идет о простой системе регистрации спорных ситуаций.

Чтобы узнать, куда следует направить основные силы и где необходимо обеспечить высокое качество продукта, важно понять, какие области составляют смысловое ядро и являются ключевыми для успеха приложения. Это знание приобретает-

ся в ходе рабочих встреч по переработке знаний, где вы совместно с экспертами в предметной области выявляете наиболее важные аспекты разрабатываемого продукта.

Смысловое ядро

Чтобы понять, что является смысловым ядром для продукта, разрабатываемого по заказу бизнеса, нужно ответить самому себе на несколько вопросов. Какие части продукта обеспечат ему успех? Почему эти части так важны? И почему они не могут быть куплены на стороне? Другими словами, что оправдывает разработку вашей системы?

Ключевые подсистемы (core parts) представляют то основополагающее конкурентное преимущество, которое может получить ваша компания благодаря запуску этого программного обеспечения. Что именно является смысловым ядром, очевидно не всегда.

Если в неспециализированных областях реализация может быть позаимствована извне, так что разработка не потребует больших усилий, то в случае смыслового ядра ситуация полностью противоположна. Разработка в смысловом ядре требует привлечения лучших разработчиков — ваших «спецназовцев». На смысловое ядро может приходиться далеко не самая большая доля в технологиях вашей компании, но оно определенно потребует самых больших вложений.

Границы смыслового ядра могут изменяться с течением времени. В случае вашего успеха конкуренты позаимствуют у вас какие-то идеи, и без постоянного развития смыслового ядра ваш бизнес может утратить лидерское положение. Очень важно, чтобы команда разработчиков уяснила эту идею и следила за тем, чтобы смысловое ядро оставалось актуальным как для программного продукта, так и для бизнеса.

СМЫСЛОВОЕ ЯДРО ПРОЕКТА POTTERMORE.COM

Сайт Pottermore.com — единственное место в Сети, где можно купить электронные экземпляры книг о *Гарри Поттере*. Подобно любому другому сайту электронной коммерции, он позволяет просматривать предлагаемые книги, откладывая их в корзину и оформлять заказ. Смысловым ядром сайта Pottermore является не то, что видит покупатель, а то, что ему как раз не видно. Книги, распространяемые сайтом Pottermore, не защищены средствами DRM (<http://www.futurebook.net/content/pottermore-finally-delivers-harry-potter-e-books-arrive>), но в них есть «водяные знаки». Эти невидимые метки позволяют следить за незаконным размещением купленных книг в Сети. Смысловым ядром системы Pottermore является подобласть, которая позволяет использовать технологию «водяных знаков» для предотвращения незаконного распространения книг, не ограничивая при этом добропорядочного пользователя (покупатель может скопировать книгу на любое свое устройство). Именно эта особенность наиболее важна для бизнеса, именно она отличает Pottermore от других продавцов электронных книг и именно из-за нее было принято решение не пользоваться другими торговыми площадками вроде iTunes, а создать свою систему.

Рассматривайте смысловое ядро как продукт, а не как проект

При разработке программного обеспечения для сложного смыслового ядра в сознании и разработчиков, и представителей бизнеса должен произойти фундаментальный сдвиг: они должны сосредоточиться на продукте как таковом и перестать рассматривать его как отдельный проект. Зачастую разработка программного обеспечения для бизнес-продукта не прекращается никогда — продукт последовательно переживает периоды интенсивного улучшения и расширения. Инвестиции в программный продукт продолжаются до тех пор, пока он представляет собой ценность для бизнеса и пока дальнейшие изменения ведут к его расширению и совершенствованию.

Ваш продукт — это последовательность дополнений, исправлений и улучшений. Бизнес, как и разработка, часто продвигается вперед итерациями. Хорошая идея становится лучше в процессе дальнейшей детализации и совершенствования. Поймите, в чем состоит ценность продукта, над которым вы работаете, и какую отдачу (ROI) приносят компании инвестиции в него. Поговорите со своими спонсорами из бизнеса о будущем продукта, чтобы более точно направлять свои усилия по созданию кода; узнайте, что для них важно.

Нередко к программному обеспечению для смыслового ядра бизнеса не относятся как к продукту, требующему заботы и внимания. Во главу угла ставится быстрый вывод на рынок — в ущерб качеству программного обеспечения и долгосрочным инвестициям. Слишком много времени уделяется размышлениям о ходе проекта и надвигающихся сроках сдачи и слишком мало — заботе о будущем продукта. В результате получается программный код, который сложно сопровождать и расширять, — и продукт скатывается к антишаблону «Большой ком грязи», описанному в главе 1 «Что такое предметно-ориентированное проектирование?».

На другой чаше весов, однако, находится отложенная дата выпуска продукта, которую часто не получается отстоять, когда на сроки запуска программного обеспечения тесно завязаны какие-то другие интересы бизнеса. Чтобы выйти из этого затруднительного положения, можно пересмотреть и исключить некоторые функциональные возможности, сохранив тем самым высокий уровень качества продукта и уложившись в сроки разработки. Однако для этого вам необходимо видеть и разделять образ продукта и конечную цель, ради которой создается программное обеспечение, — тогда вы сумеете включить в первую версию продукта действительно самые важные функции и особенности и тем самым обеспечить его высокую ценность для бизнеса и соответствие ожиданиям.

Неспециализированные области

Неспециализированная область — это такая подобласть, которая есть во многих крупных системах для бизнеса. В качестве примера неспециализированной области можно привести службу отправки электронной почты, пакет управления учетными записями или инструментарий формирования отчетов. Эти подобласти не являются основными для бизнеса, но без них работа не будет выполняться должным образом. Поскольку такие подобласти не входят в смысловое ядро и не

обеспечивают ключевых конкурентных преимуществ, нет смысла тратить слишком много сил и средств на их разработку. Программное обеспечение для неспециализированных областей стоит поискать на стороне либо поручить его создание менее опытным разработчикам, высвободив тем самым более мощные ресурсы для работы над смысловым ядром.

Однако имейте в виду, что в проектах, ориентированных на коммуникацию и электронную рассылку коммерческих предложений с ограниченным сроком действия, таких как Groupon или Wowcher, смысловым ядром может оказаться именно сложная система управления отношениями с клиентами или рассылки сообщений по электронной почте. То, что является смысловым ядром для одного бизнеса, может быть неспециализированной областью для другого.

Поддерживающие области

Остальные подобласти в системе определяются как поддерживающие области. Это подобласти, которые обеспечивают поддержку смыслового ядра, хотя и не играют ведущей роли в системе. Например, для Amazon поддерживающей областью будут функции, позволяющие клиентам просматривать каталог продуктов. Эта функция не является для компании Amazon определяющей и ничем не отличается от аналогичных функций на других сайтах электронной коммерции, но она поддерживает трассировку перемещений пользователей, снабжая информацией механизм выработки рекомендаций.

Как и в случае неспециализированных областей, для поддерживающих областей следует по возможности найти стороннее программное обеспечение. Если это не удалось, не тратьте на разработку поддерживающих систем много сил: они должны работать, но не требуют чересчур пристального внимания. Важно отметить, что для поддерживающей области не всегда требуется техническое решение — возможно, потребности бизнеса можно покрыть ручным процессом, направив силы разработчиков на смысловое ядро.

Как деление на подобласти формирует контур решения

Внутри каждой подобласти можно создать свою модель. На рис. 3.5 показано, каким образом сайт электронных торгов был разделен на два физических приложения. Область урегулирования споров в данном случае покрывается возможностями готового стороннего пакета, а смысловое ядро и поддерживающие области реализуются в виде собственного веб-приложения.

Не все части системы требуют детальной проработки

Внутри каждой подобласти должна существовать модель, которая представляет предметную логику и бизнес-правила соответствующей части системы. Не все такие модели будут иметь одинаково высокое качество проработки. Хорошо ра-

зобравшись в различных подобластях, составляющих вашу систему, вы сможете разумно распределить силы и применить шаблоны проектирования на основе модели к тем частям, где это даст наибольшую отдачу.

Не тратьте время и силы на рефакторинг всего программного кода — сосредоточьтесь прежде всего на смысловом ядре. Если программный код, реализующий неспециализированные и поддерживающие области, окажется в конечном итоге неряшливым, но работоспособным, оставьте его в покое. Помните, что лучшее — враг хорошего. Наличие небольших «комков грязи» вполне допустимо, если они находятся в четко очерченных границах. Совершенство — это иллюзия. Совершенства следует требовать только от кода, составляющего смысловое ядро. Бизнесу не нужно высокое качество кода в тех частях системы, которые необходимы, но не являются ключевыми и вряд ли будут развиваться впоследствии.



Рис. 3.5. Как решение отображается на подобласти системы электронных торгов

Уделяйте больше внимания ясности границ, а не совершенству моделей

Шаблон «Большой ком грязи» — самый популярный архитектурный шаблон. В крупных программных системах, развивавшихся в течение длительного времени, почти наверняка есть области, реализованные далеко не идеально. Если в вашем приложении имеются части, напоминающие шаблон ВВоМ, то лучшее, что вы можете сделать, — очертить их границы, чтобы «грязь» не расплзлась в новые области приложения. На рис. 3.6 показано пространство решения приложения, в котором явно определены границы между унаследованными областями ВВоМ и новыми моделями. Для предотвращения взаимопроникновения и смешивания моделей можно использовать предохранительный слой.

ПРЕДОХРАНИТЕЛЬНЫЙ СЛОЙ

Предохранительный слой (anticorruption layer) пропускает через себя взаимодействие с унаследованным или сторонним программным кодом, тем самым защищая целостность ограниченных контекстов. Он осуществляет преобразование между представлениями контекстов, обеспечивая целостность нового программного кода и не давая ему превратиться в «большой ком грязи». Подробнее о шаблоне предохранительного слоя рассказывается в главе 7 «Карты контекстов».

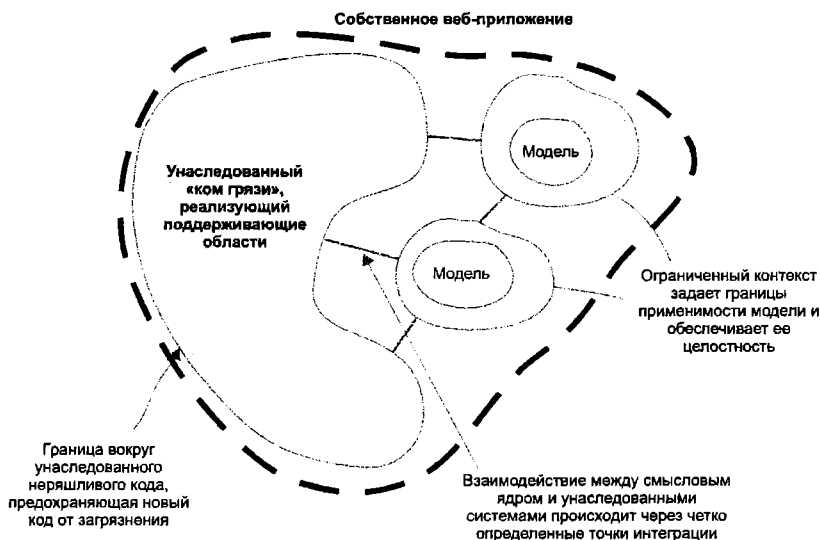


Рис. 3.6. Работа с унаследованным кодом

Реализация смыслового ядра не обязана быть идеальной с самого начала

В идеальном мире качество программного обеспечения всегда стояло бы в списке ваших приоритетов на первом месте; однако в реальной жизни важно быть более прагматичными. Иногда смысловое ядро новой системы может быть первым на рынке, иногда представители бизнеса не в состоянии сказать, удачна ли та или иная идея и является ли она ключевой для успеха системы. В таких ситуациях для бизнеса важно быстро опробовать идею и оценить ее пригодность до того, как на ее воплощение будут затрачены существенные ресурсы.

Первая версия продукта, перспективы которого пока непонятны бизнесу, не обязана быть хорошо проработанной. Это нормально, потому что у бизнеса нет уверенности в том, что продукт получит дальнейшее развитие, и разработчики должны понимать, почему бизнес предпочитает быстрые результаты гибкому дизайну.

Но если продукт окажется успешным и будет принято решение о долгосрочных инвестициях в разработку его программного обеспечения, вам придется провести рефакторинг кода, чтобы обеспечить последующее развитие; в противном случае технические недоработки, допущенные вначале ради быстрого запуска («технический долг»), превратятся в проблему.

Создавайте подобласти с прицелом на замену, а не на повторное использование

Разрабатывая модели в отдельных подобластях, старайтесь конструировать их изолированными друг от друга, чтобы облегчить их будущую замену. Отделяйте их от других моделей, унаследованного программного кода и сторонних служб, задавая четкие границы. Программируя с прицелом на замену, а не на повторное использование, вы сможете создать поддерживающие области приемлемого качества, не тратя силы на их доведение до совершенства. В будущем их можно будет заменить стандартными сторонними решениями или переписать с учетом изменившихся потребностей бизнеса.

А что, если смысловое ядро отсутствует?

У бизнес-заказчика может быть множество причин для разработки своего программного обеспечения вместо приобретения готового. Если вы способны произвести нужный продукт дешевле, быстрее или лучше, это может послужить веским доводом в пользу разработки своими силами. Если окажется, что создаваемое вами программное обеспечение стандартно или просто поддерживает другие приложения компании, а как следствие, смысловое ядро в нем отсутствует, не пытайтесь применять в проекте все приемы и принципы DDD. Стратегические шаблоны DDD будут вам по-прежнему полезны, а вот применение тактических шаблонов проектирования на основе модели может стать пустой тратой сил. О том, когда стоит и когда не стоит применять шаблоны проектирования на основе модели, подробнее рассказывается в главе 9 «Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование».

Ключевые идеи

- Дистилляция используется, чтобы разбить большую предметную область на подобласти и выделить смысловое ядро, поддерживающие области и неспециализированные области.
- Дистилляция помогает уменьшить сложность пространства задачи.
- Внимание и силы команды следует направить прежде всего на смысловое ядро. Работать с ним должны самые опытные разработчики.
- Смысловое ядро — это та причина, по которой вы создаете собственное программное обеспечение.

- Для неспециализированных и поддерживающих областей рассмотрите такие варианты, как привлечение сторонних разработчиков, покупка готового программного обеспечения или разработка силами менее опытных специалистов.
- Обзор предметной области — документ, который помогает сформировать единые представления о том, что является ключевым для успеха продукта. Чтобы составить обзор предметной области, прибегните к помощи экспертов в предметной области, изучите документацию по запуску проекта, ознакомьтесь с презентациями бизнес-стратегии компании.
- Будьте готовы к замене модели смыслового ядра по мере углубления ваших представлений о задаче. Не привязывайтесь чрезмерно к какому-то одному решению — смысловое ядро может меняться со временем.
- Не все части системы требуют детальной проработки. Сосредоточьте основные усилия на смысловом ядре. Для унаследованных систем задайте защитные границы, чтобы исключить смешивание нового программного кода со старым.

4

Проектирование на основе модели

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Определение предметной модели
- Связывание программной модели с аналитической моделью посредством единого языка
- Важность единого языка
- Как совместно формировать единый язык для улучшения коммуникации
- Советы по созданию эффективных предметных моделей
- Когда следует применять приемы проектирования на основе модели

Когда достигнуто глубокое и разделяемое всеми участниками проекта понимание предметной области и ключевых аспектов, важных для успеха приложения, можно сосредоточиться на пространстве решения. Однако при этом важно воплотить в программном коде ту аналитическую модель, которая была создана в ходе встреч по переработке знаний, то есть модель, понятную представителям бизнеса. Разделение программной и аналитической моделей, характерное для традиционных процессов создания программного обеспечения, приводит к тому, что фактическая реализация слабо напоминает исходный проект из-за ограничений соответствующего технического решения и переосмысления задачи по ходу дела. В предметно-ориентированном проектировании упор делается на создании единой модели, которая выступает в роли аналитической модели, понятной бизнесу, и реализуется в программном коде с использованием тех же самых терминов и понятий.

Этот процесс известен как «проектирование на основе модели» (Model-Driven Design). В том, что касается увязки технической реализации модели с аналитической моделью и поддержки их взаимосогласованности на протяжении всего жизненного цикла системы, он существенно опирается на единый язык. В этой главе мы не только подробно обсудим проектирование на основе модели и единый язык, но поговорим также о шаблонах, помогающих создавать эффективные предметные модели, и о тех сценариях, для которых следует использовать проектирование на основе модели.

Что такое предметная модель?

Модель предметной области, или предметная модель (domain model), является центральным элементом предметно-ориентированного проектирования, как показано на рис. 4.1. Она возникает как аналитическая модель в ходе совместной деятельности команды разработчиков и бизнес-специалистов на встречах по переработке знаний. При этом предметная модель отражает не реальность предметной области во всей полноте, а упрощенное представление о ней, выстроенное таким образом, чтобы соответствовать требованиям бизнес-сценариев использования. Она описывается посредством общего для команды языка и нарисованных командой схем. Реализация модели в программном коде сохраняет связь с аналитической моделью благодаря тому, что использует тот же общий язык. Ее ценность обусловлена ее способностью представить сложную логику и правила предметной области таким образом, чтобы обеспечить выполнение бизнес-сценариев использования. Модель включает в себя только то, что имеет отношение к решению задач в контексте создаваемого приложения. Чтобы оставаться правильной и полезной, она должна постоянно развиваться, следуя за изменениями в бизнесе.

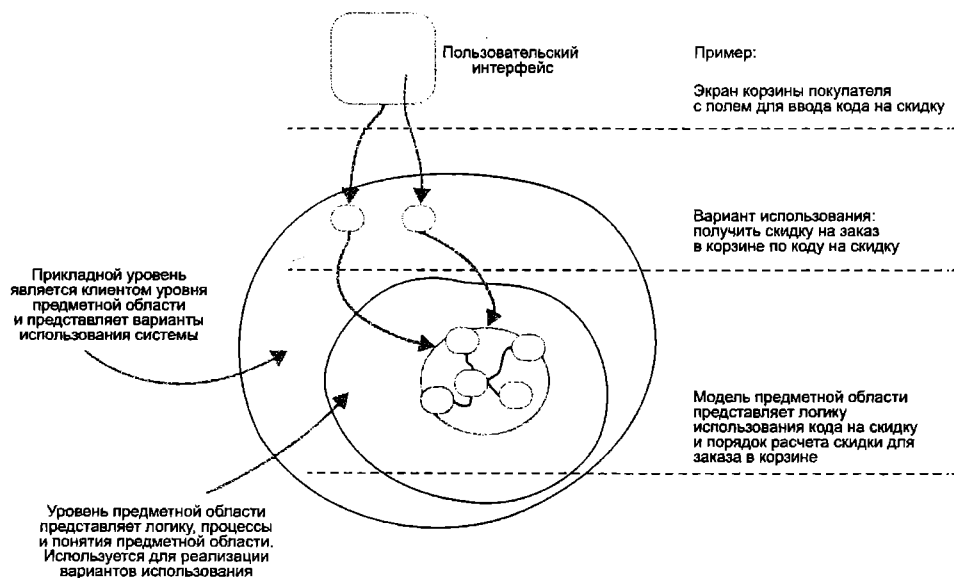


Рис. 4.1. Роль предметной модели

Предметная область и ее модель

Предметная область представляет окружение задачи, над решением которой вы работаете. Это реальная ситуация во всей ее полноте. Предметная модель, напротив, является абстракцией предметной области, облеченной в форму программного кода, и отражает некоторое представление о реальности, а не саму реальность. Рису-

нок 4.2 подчеркивает это различие. Ценность предметной модели определяется не тем, насколько точно она отображает реальность, а ее способностью представлять сложную логику и правила предметной области в виде, пригодном для решения бизнес-задач. Пространство, в котором она существует, также более абстрактно: это язык, на котором говорит команда, и нарисованные командой диаграммы. Модель строится разработчиками в сотрудничестве с бизнес-специалистами. Предметная модель существует лишь затем, чтобы помогать в решении задач. Для того чтобы эффективно выступать в этом качестве, она должна быть ясной и свободной от технических сложностей — в этом случае бизнес-специалисты и команда разработчиков смогут плодотворно сотрудничать над созданием ее архитектуры.



Рис. 4.2. Предметная область и ее модель

Аналитическая модель

Аналитическая модель (analysis model), иногда называемая также бизнес-моделью, — это набор артефактов, описывающих модель системы. В роли таких артефактов может выступать что угодно, от набросков на сигаретной пачке до неформальных UML-моделей. Задача аналитической модели — помочь разработчикам и бизнес-пользователям разобраться в предметной области. Аналитическая модель не является планом технической реализации.

Программная модель

DDD отнюдь не призывает к отказу от аналитической модели: модель, описывающая систему, обладает высокой ценностью. Наоборот, DDD подчеркивает необходимость сохранять тесную взаимосвязь между программной моделью (реализацией) и аналитической моделью (проектом). Эта взаимосвязь достигается за счет использования единого языка для описания обеих моделей, как показано на рис. 4.3. В идеале хорошо бы иметь единую модель, полезную и при проектирова-

нии, и при реализации. Чтобы добиться этого, чрезвычайно важно поддерживать программную модель сосредоточенной вокруг предметной области и свободной от технических сложностей. Аналитическая модель, в свою очередь, должна быть пригодной для реализации — не слишком абстрактной или верхнеуровневой.

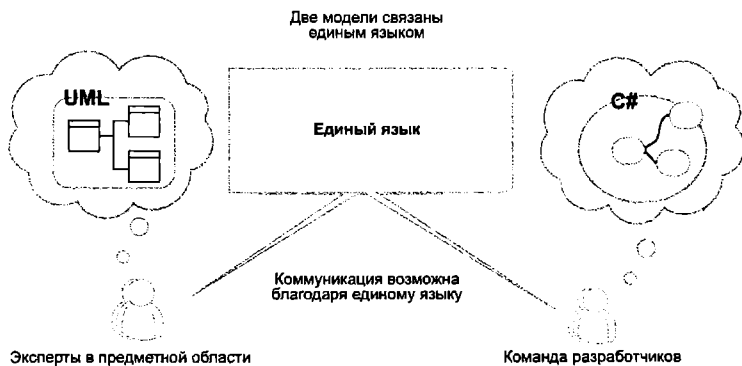


Рис. 4.3. Связь между программной и аналитической моделями

Программная модель служит основным представлением предметной модели

Программная модель (code model) — это реализация аналитической модели; она позволяет проверить предположения бизнес-специалистов и быстро выявить нестыковки в аналитической модели. Если при создании программной модели обнаруживаются сложности, не укладывающиеся в логику модели, команда разработчиков в сотрудничестве с экспертами в предметной области должна найти решение. Эти изменения, внесенные в программную модель, отражаются в аналитической модели путем изменения описания рабочих процессов и правил, которые прежде могли не вызывать вопросов. Аналогично, любые изменения в аналитической модели с позиций бизнеса находят свое отражение в программной модели. Программная и аналитическая модели должны оставаться согласованными. Программный код — это модель; программный код — это истина.

Проектирование на основе модели

Проектирование на основе модели — это процесс увязывания аналитической модели с ее программной реализацией, который обеспечивает их взаимную согласованность и полезность в ходе их развития. Он служит для проверки и доказательства практической пригодности модели, поскольку нет никакого смысла в тщательной проработке модели, если ее нельзя реализовать на практике. Разница между проектированием на основе модели и предметно-ориентированным проектированием заключается в том, что в центре внимания первого подхода находятся прежде всего практическая реализация и те ограничения, которые могут

потребовать изменений в изначальной модели, тогда как второй уделяет основное внимание языку, совместной работе и переработке знаний. Эти два подхода прекрасно дополняют друг друга; проектирование на основе модели позволяет встроить предметные знания и общий язык в программную модель таким образом, чтобы она отражала язык и ментальные модели бизнес-специалистов. Корректность моделей с той и другой стороны открывает возможности для плодотворного сотрудничества бизнес-специалистов и разработчиков в процессе решения задач. Более глубокое понимание, достигнутое при помощи любой из моделей, становится достоянием всех членов объединенной команды и увеличивает объем накопленных знаний, что облегчает процесс поиска решений и вносит ясность в коммуникацию между разработчиками и специалистами со стороны бизнеса.

Проблемы, возникающие при заблаговременном проектировании

Исторически выявление требований к программным системам рассматривалось как деятельность, которая может протекать задолго до написания программного кода. Специалисты со стороны бизнеса общаются с бизнес-аналитиками, те, в свою очередь, взаимодействуют с архитекторами, а последние разрабатывают аналитическую модель на основе всей собранной информации о предметной области. Затем эта аналитическая модель вместе со схемами и диаграммами рабочих процессов передается разработчикам, которые создают систему.

Приступив к реализации аналитической модели в программном коде, разработчики часто обнаруживают нестыковки между верхнеуровневыми артефактами, которые подготовлены архитекторами, и реальностью создаваемой системы. Однако механизмы обратной связи, которые позволили бы разработчикам обсудить изменения в аналитической модели с архитекторами и представителями бизнеса, на этом этапе часто отсутствуют. В итоге разработчики отступают от аналитической модели и нередко упускают в получившейся реализации важные термины и понятия, которые могли бы помочь разобраться в предметной области более глубоко.

По мере того как команда разработчиков все дальше отходит от аналитической модели, та становится все менее и менее полезной. Перенос фокус внимания с понятий бизнеса на уводящие в сторону технические моменты, разработчики теряют из виду важные элементы, ключевые для понимания модели. Работа подходит к завершению, но в программном коде практически не остается следов исходной аналитической модели. Бизнес по-прежнему уверен в правильности исходной модели и не подозревает об отступлениях в коде.

На рис. 4.4 показано, как может нарастать разрыв между аналитической и программной моделями, когда команда разработчиков не принимает активного участия в переработке знаний.

Проблема проявляется в тот момент, когда становится сложно реализовать в программном коде последующие улучшения. Источником этих сложностей служит то, что бизнес-специалисты и разработчики пользуются разными моделями бизнеса. Программный код оторван от бизнес-процессов и не подкреплён в должной мере знаниями из предметной области.

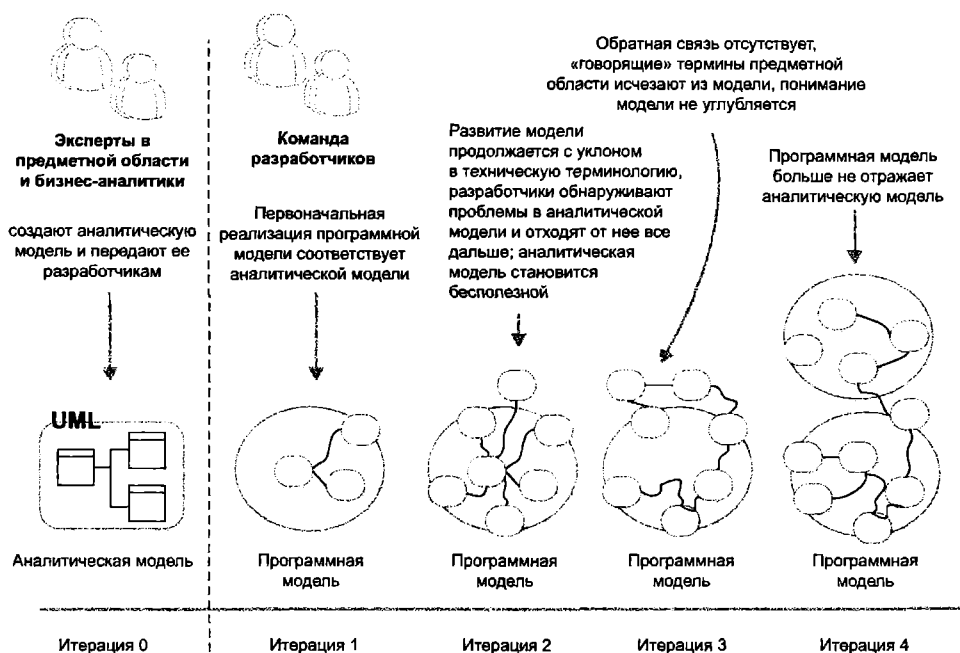


Рис. 4.4. Проблемы, возникающие при заблаговременном проектировании

Командное моделирование

Метод выявления требований к системе и прояснения существующих процессов, предлагаемый в рамках предметно-ориентированного проектирования, в большей мере направлен на сотрудничество. Упор делается на том, что пространство задачи обсуждается всей командой, включая бизнес-специалистов и архитекторов (постольку, поскольку они вовлечены в написание программного кода). Обсуждаться могут любые документы или унаследованный код, так или иначе связанные с создаваемой системой. В основе встреч по совместной переработке знаний лежит идея о том, что разработчики, тестировщики, бизнес-аналитики, архитекторы и специалисты со стороны бизнеса должны действовать как единая команда. Разработчикам и тестировщикам это поможет разобраться в предметной терминологии и понять сложную логику предметной области. Бизнес-специалисты, в свою очередь, получают опыт моделирования и осваивают на практике соответствующие приемы. Понимая, в чем состоит суть процесса моделирования, специалисты со стороны бизнеса могут самостоятельно создавать модели и проверять результаты проектирования с участием разработчиков.

Благодаря обмену информацией бизнес-специалисты вносят свой вклад в проектирование приложения, а разработчики глубже вникают в специфику предметной области. Спустя некоторое время разработчики и бизнес-специалисты соберут достаточно информации, чтобы построить первую модель предметной области. Эта первоначальная модель проверяется на предметных сценариях — реальных

задачах из предметной области, которые позволяют проверить пригодность модели. Моделирование вслух с использованием терминов и языка модели также помогает проверить ранние результаты проектирования.

Важным аспектом совместного моделирования является постоянная обратная связь, которую разработчики получают от бизнес-специалистов. Она позволяет команде вовремя обнаруживать важные понятия и четче понимать, что является несущественным и может быть исключено из модели. Результаты встреч по переработке знаний воплощаются в простые абстракции, проясняющие сложные понятия предметной области и делающие модель более выразительной.

Затем модель переносится в программный код — появляются первые версии приложения, которые дают разработчикам и бизнес-специалистам возможность получить быструю обратную связь. Обратная связь становится стимулом для более глубокого понимания задачи, которое отражается в коде и в аналитической модели, как показано на рис. 4.5.

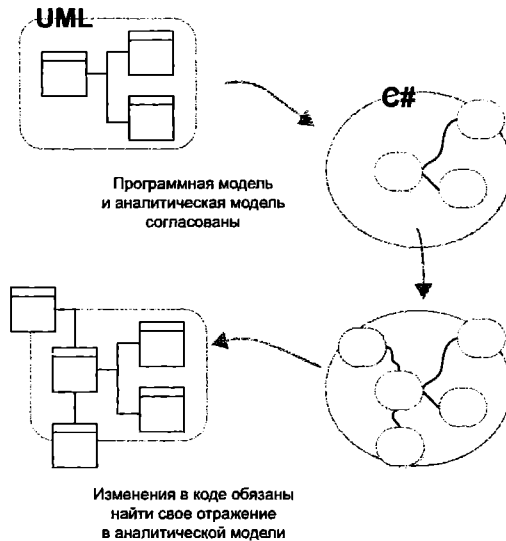


Рис. 4.5. Поддержка согласованности программной и аналитической моделей

В ходе каждой итерации разработчики могут обнаружить в модели какие-то части, которые представлялись полезными и вроде бы должны были решать свою задачу, но на поверку в процессе реализации потребовали изменений. Новообретенное знание передается специалистам со стороны бизнеса, чтобы те могли предложить пояснения и уточнить свои представления о предметной области. В этом процессе программная и аналитическая модели составляют единое целое, и изменение в одной из них приводит к изменениям в другой.

На рис. 4.6 показано, как аналитическая и программная модели поддерживаются в согласованном состоянии и продолжают развиваться как единое целое на протяжении всей работы над продуктом.

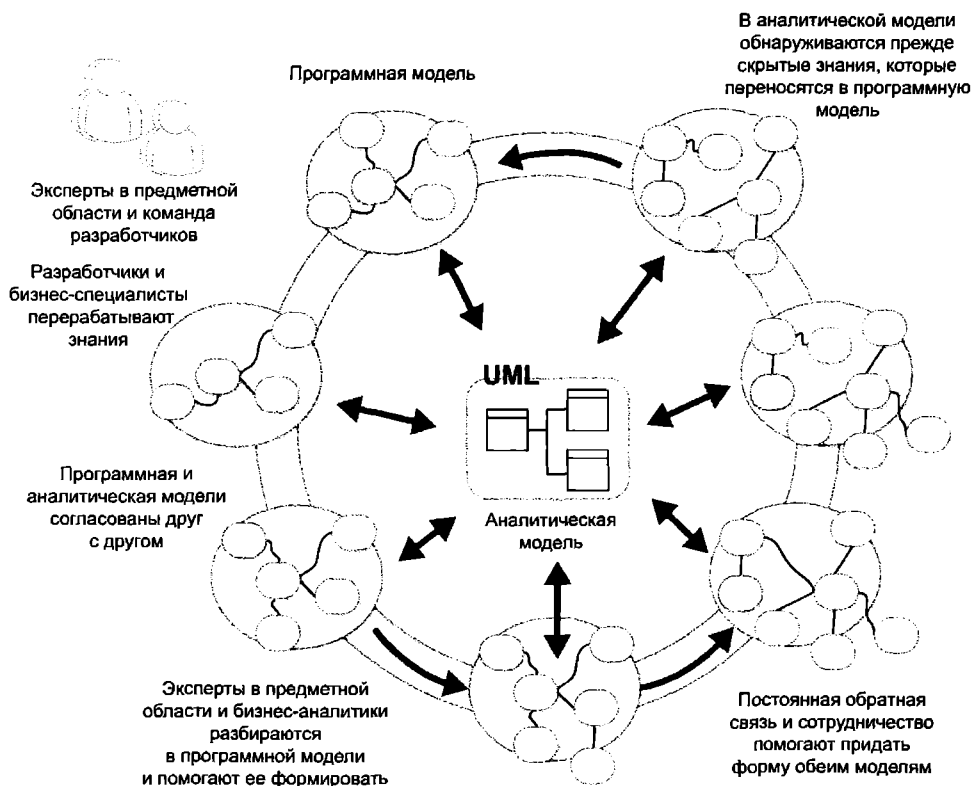


Рис. 4.6. Командное моделирование

Использование единого языка для связывания аналитической и программной моделей

Истинная ценность философии предметно-ориентированного проектирования заключена в сотрудничестве разработчиков и специалистов, направленном на то, чтобы глубже понять предметную область. Созданный в результате программный код — лишь артефакт этого процесса, хотя и немаловажный. Развитие понимания требует эффективной коммуникации. Именно создание единого языка (Ubiquitous Language, UL) помогает достичь глубокого понимания, которое сохранится и тогда, когда программный код будет переписан и даже полностью заменен.

Единый язык облегчает структурирование как ментальной, так и программной модели. Он формирует у всех участников проекта единые представления и благодаря этому становится четким и недвусмысленным. Он обеспечивает также ясность и непротиворечивость формулировок. В конечном счете он находит свое

отражение в программном коде, однако в его создании важную роль играют также устная речь, схематические наброски и документация. Единый язык постоянно исследуется, проверяется и шлифуется по мере появления новых сведений и углубления знаний.

Язык переживет вашу программу

Ценность создания единого языка простирается за пределы его применения для разработки конкретного продукта, над которым вы работаете в данный момент. Он помогает явным образом описать, чем занимается бизнес, глубже вникнуть в суть процессов и их логики и упростить деловую коммуникацию.

Язык бизнеса

Недавно мне довелось сопровождать жену в магазин штор. «Плиссировка», «люверсы», «флизелин» — в предметной области изготовителей штор все эти термины имеют четкий смысл. Работники магазина могли бы часами описывать, что они имеют в виду, и все же оставаться не до конца понятыми. Но благодаря использованию терминологии из предметной области магазина штор им удается передавать свои мысли друг другу быстро и ясно, причем любой, кто разбирается в этой предметной области, без труда поймет их.

То же относится к плотникам, биржевым трейдерам, военнослужащим и представителям практически любой предметной области, которая может прийти в голову. В каждой профессии есть свои термины и понятия, обозначающие что-то весьма специфическое. Этот тайный единый язык позволяет охватить сложные темы в кратком и содержательном диалоге, не пускаясь в пространные объяснения. Разработчикам жизненно необходимо научиться понимать этот язык и совместно с бизнес-специалистами работать над его формированием. Термины и понятия единого языка используются в ходе взаимодействия с другими членами команды, включая экспертов в предметной области. Те же самые термины и понятия служат для именования классов, методов и пространств имен в программном коде.

Перевод между языком разработчиков и бизнес-языком

Язык бизнеса — это богатый диалект с «говорящей» терминологией, имеющей глубокий смысл. Однако если разработчики не будут привлекать экспертов в предметной области, чтобы научиться понимать этот язык и использовать его в программной реализации, они лишат себя многих его преимуществ. В этом случае разработчики создадут для предметной области свой собственный язык и набор абстракций. Без общей модели и единого языка эффективное общение разработчиков с экспертами в предметной области будет затруднено и потребует постоянного перевода с языка предметных понятий на язык технических понятий. Такой перевод отнимает время и чреват ошибками. Когда команда, занятая созданием программного кода, использует модель, отличающуюся от модели экспертов в предметной области, могут быть потеряны критически важные предметные нюансы. Кроме того, те сложности, которые возникают у разработчиков в процессе реализации и требуют пространных

и путаных объяснений, зачастую можно было бы легко обойти, имея более ясное представление о предметной области и более эффективный способ общения.

Рисунок 4.7 иллюстрирует те трудности, которые возникают при общении с экспертом в предметной области у разработчика, мыслящего в рамках другой модели. Разработчик сосредоточился на технических абстракциях в программном коде, шаблонах и принципах проектирования, тогда как в центре внимания эксперта находятся бизнес-процессы.

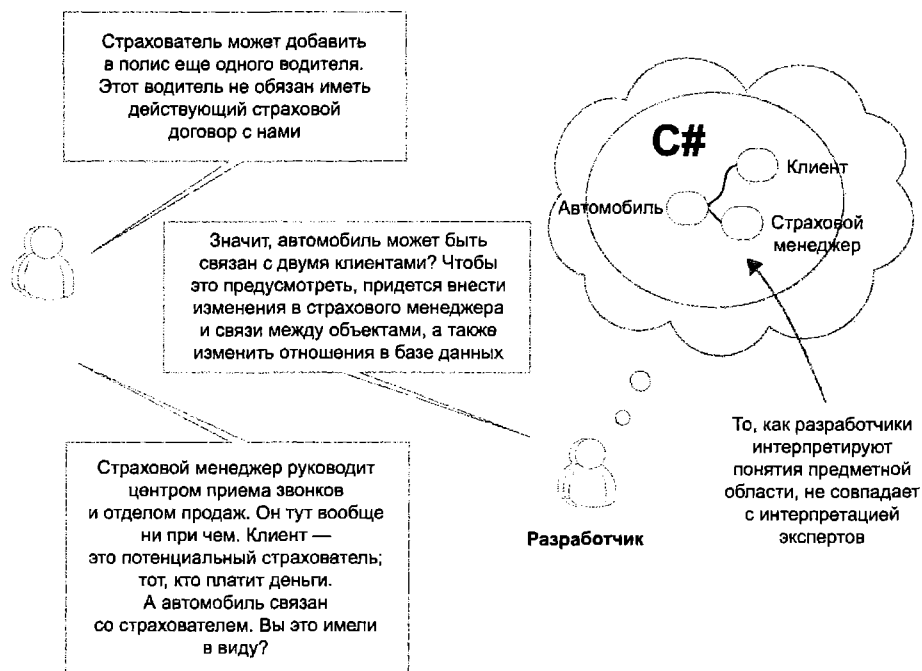


Рис. 4.7. Цена перевода понятий с одного языка на другой

Разработчики должны мыслить не в технических терминах и понятиях, а в предметных, чтобы избежать необходимости перевода с бизнес-жаргона на технический жаргон и обратно. Если при переводе сложной логики или бизнес-процесса разработчики что-то интерпретировали неверно, вероятность появления ошибок в программном коде значительно возрастет.

Совместная работа над созданием единого языка

Тот богатый язык, с помощью которого бизнес-специалисты описывают свою деятельность, — один из ингредиентов единого языка. Однако при создании модели предметной области и в ходе ее реализации в программном коде вам может по-

требоваться ввести новые термины и понятия. Как и в жаргоне ИТ-сообщества, в жаргоне бизнеса некоторые термины могут оказаться слишком общими. Для реализации модели в программном коде разработчикам и экспертам в предметной области понадобится создать новые термины и явным образом очертить значение существующих терминов.

По мере реализации модели в коде могут обнаружиться новые понятия — часто в форме набора объектов или логики поведения, которым нужно дать имя. Термины, возникшие в результате таких открытий, нужно передать на уточнение и утверждение экспертам в предметной области.

Разработчики должны не только подхватывать и осваивать явные термины и понятия, используемые бизнесом, но также вместе с экспертами давать четкие определения подразумеваемым и неявным понятиям, для которых может не быть устоявшейся терминологии. Названия этим понятиям должны даваться всей объединенной командой и включаться в совместно используемый единый язык. Команде может понадобиться также создать термины для понятий, отсутствующих в предметной области, но обнаруженных в процессе разработки программной модели и требующих обозначения.

Члены команды должны общаться друг с другом на едином языке. Разработчики используют его в программном коде, а эксперты в предметной области пользуются им при общении с командой разработчиков. Общий язык избавляет от необходимости переводить с языка бизнеса на технический язык разработчиков и обратно. Он устраняет также неоднозначности и возможности неверного толкования, поскольку все понимают, какой смысл вкладывается в то или иное понятие.

Единый язык обязан быть ясным и кратким. Он не должен содержать технических терминов из области программирования, чтобы они не затеняли собой бизнес-понятия. Аналогичным образом не следует замусоривать его теми предметными терминами, которые никак не связаны с создаваемым продуктом.

Шлифовка языка на конкретных примерах

Как отмечалось в главе 2 «Дистилляция предметной области задачи», чтобы лучше понять ту предметную область, с которой вы работаете, полезно исследовать происходящее в ней на конкретных примерах. Взятые из практики сценарии помогут связать процессы и понятия в предметной области в единое целое. Однако при этом важно вскрыть истинное назначение бизнес-процесса, не отвлекаясь на его реализацию. Используйте для описания только термины из бизнеса, не сбивайтесь на техническую терминологию.

В следующем примере пользователь со стороны бизнеса описывает, каким образом клиент сайта электронной коммерции оформляет замещающий заказ взамен недоставленного:

Если клиент не получил свой товар, он может бесплатно оформить новый заказ. Для этого клиент должен войти под своей учетной записью и щелкнуть на кнопке с надписью Я не получил свой заказ. Если клиент отмечен как уже получивший бесплатный заказ, он не сможет оформить еще один без обращения в отдел по работе

с клиентами. В противном случае мы отправим клиенту заказ бесплатно и поставим в базе данных флажок, что этот клиент уже заявил о потере заказа. Затем мы свяжемся с курьером, чтобы выяснить, можем ли мы истребовать стоимость потерянного заказа.

Обратите внимание, что в этом описании внимание пользователя сосредоточено не на самом процессе, а на деталях реализации. С точки зрения знаний о предметной области или о бизнес-процессе приведенное ниже предложение не дает никаких ключей и в этом смысле не имеет какой-либо ценности:

Для этого клиент должен войти под своей учетной записью и щелкнуть на кнопке с надписью Я не получил свой заказ.

В предложении, которое следует за ним, пользователь пытается предвосхитить конкретную реализацию бизнес-правила. Иногда эксперты обладают определенным опытом работы с базами данных и могут доходить даже до составления схем данных. Здесь также нет ничего, что было бы существенно для понимания предметной области:

Если клиент отмечен как уже получивший бесплатный заказ, он не сможет оформить еще один без обращения в отдел по работе с клиентами.

С таким набором требований разработчики, не интересующиеся предметной областью, могут просто реализовать то, что было сказано, и в итоге получить скверную модель, не отражающую понятия и правила предметной области. Результатом может стать неверная трактовка термина «отмеченный клиент»: за ним в действительности может скрываться нечто большее, чем просто «галочка» в базе данных и возможное начало отдельного потока бизнес-операций. Буквально реализовав услышанное без понимания предметной области и истинного назначения тех или иных функций, разработчики вряд ли будут удовлетворены последствиями.

Учите экспертов в предметной области сосредотачиваться на задаче и не переходить к ее решению

Обучение и сотрудничество помогут специалистам со стороны бизнеса удерживать в центре внимания процесс, а не реализацию, пространство задачи, а не пространство ее решения. Вот как можно переписать предыдущие требования на языке предметной области, поставив во главу угла сам бизнес и протекающие в нем процессы:

Если вы не получили заказ, то можете отправить уведомление о недоставленном заказе. Если это ваша первая претензия, автоматически будет создан замещающий заказ. Если прежде вы уже оставляли такую жалобу, будет открыто дело о претензии и назначен ответственный из отдела по работе с клиентами для его расследования. В любом случае будет открыто дело о возмещении стоимости потерянного заказа, которое будет направлено курьеру с подробной информацией о недоставленном отправлении.

Это описание выявляет множество важных понятий предметной области, упущенных ранее. Переписанные таким образом требования вводят несколько новых терминов в единый язык, и терминология предметной области становится намно-

го более понятной. В действительности во втором описании вообще отсутствует понятие клиента — оно выстроено исключительно вокруг терминов, имеющих прямое отношение к процессу.

Помните: эксперты в предметной области плохо разбираются в технической терминологии или не понимают ее вовсе. Добивайтесь того, чтобы примеры описывали сам бизнес, а если эксперты пытаются помочь вам, перескакивая к подробностям реализации, вежливо возвращайте их к обсуждению вопросов о том, «что» и «почему» должна делать система, и предлагайте положиться на вас в том, «как» это будет делаться.

Эффективные приемы формирования языка

Ниже перечислены некоторые наиболее эффективные приемы, которые помогут вам в формировании единого языка.

- Добивайтесь лингвистической целостности. Если в программном коде используется термин, который эксперт в предметной области не произносит, необходимо согласовать этот термин с экспертом. Возможно, вы обнаружили важное понятие, которое должно быть добавлено в единый язык и осознано экспертом. Но может быть также, что вы неправильно поняли что-то сказанное экспертом, и тогда вам необходимо внести исправления в программный код, заменив термин.
- Составьте вместе с экспертами словарь предметных терминов, чтобы избежать путаницы и явным образом описать предметные понятия.
- Для каждого конкретного понятия используйте одно и то же слово. Не позволяйте экспертам или разработчикам обозначать что-либо двумя (или более) разными словами: это сбивает с толку и может означать, что речь в действительности идет о двух понятиях с разными контекстами.
- Избегайте перегруженных терминов, таких как «правила», «служба» или «менеджер». Стремитесь к ясности, даже если для этого придется быть многословными.
- Не используйте термины, имеющие конкретный смысл в разработке программного обеспечения, такие как названия шаблонов проектирования, потому что разработчики могут подразумевать под ними особенности реализации, а не поведения.
- Выбор имен играет очень важную роль. Проверяйте архитектуру своего кода, рассказывая бизнес-пользователям о классах. Будет ли им понятна фраза «в кэш передается запрос на получение имен пользователей, соответствующих регулярному выражению, чтобы определить наличие у них скидки»? Звучат ли код и заложенные в него концепции осмысленным образом, когда вы произносите их вслух? Если нет, узнайте у эксперта, как он назвал бы то или иное понятие.
- Давайте исключениям имена в терминах единого языка.
- Не используйте имена шаблонов проектирования в предметной модели. Подумайте, что может означать для пользователя слово «декоратор»? Будет ли ему понятна роль «фабрики»? Может оказаться, что специалисты уже имеют

свою трактовку понятия «адаптер»; названия шаблонов проектирования, предложенных Бандой четырех¹, могут приводить их в замешательство.

- Единый язык должен присутствовать везде, от названий пространств имен и классов до названий свойств и методов. Отталкивайтесь от этого языка при формировании структуры программного кода.
- По мере того как вы будете углублять свое понимание предметной области, будет развиваться и ваш единый язык. Чтобы учесть эти перемены, проводите рефакторинг кода, используя такие имена методов, которые лучше отражают их назначение. Если вы заметили, что начинает формироваться блок сложной логики, расскажите эксперту в предметной области, что делает ваш код, — возможно, вместе вам удастся ввести подходящее предметное понятие. Если это получилось, выделите такой блок кода в отдельную спецификацию или класс правил.

ПРОВЕРЯЙТЕ МОДЕЛЬ, ПРОГОВАРИВАЯ ЕЕ ОПИСАНИЕ ВСЛУХ

Следя за лингвистической целостностью, вы можете проверять полезность модели. Например, прислушивайтесь к разговорам о модели и обращайте внимание на те понятия в ее архитектуре, которые плохо укладываются в бизнес-сценарии. Используйте предметный язык для проверки решений.

ЧТО ТАКОЕ СПЕЦИФИКАЦИЯ?

Спецификация представляет собой бизнес-правило, которому должна соответствовать по крайней мере некоторая часть предметной модели. Спецификации можно использовать также для описания критериев запросов. Например, можно запросить все объекты, соответствующие заданной спецификации.

Как создавать эффективные предметные модели

Для решения сложных задач создаются изопренные предметные модели. Лучший способ создать эффективную предметную модель состоит в том, чтобы сосредоточиться прежде всего на самых важных для бизнеса аспектах приложения. Игнорируйте те части системы, которые всего лишь управляют данными и где

¹ Банда четырех (англ. Gang of Four, сокращенно GoF) — распространенное название группы авторов, куда входят Эрих Гамма, Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides). В 1995 году Банда четырех выпустила книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (русский перевод: изд-во «Питер», 2016), посвященную шаблонам проектирования и получившую широкую известность. — *Примеч. пер.*

большинство действий сводится к CRUD-операциям¹. Основное внимание следует уделять самым сложным частям, тем аспектам смыслового ядра, которые представляют наибольший интерес для бизнеса, а также, как правило, тем частям, которые критичны для получения прибыли или экономии средств.

ПОПАЛИ В СТУПОР ПРИ СОЗДАНИИ МОДЕЛИ?

Встряхнитесь. Устройте мозговой штурм в коде, воплощая бизнес-требования в виде классов и методов. Продолжайте моделирование — на доске, на бумаге, с коллегами и даже дома с партнером. Разминайте мозг, постоянно делая что-нибудь, неважно что, — и в конце концов модель оформится в вашем сознании. Если это не работает, попробуйте посидеть в созерцательной тишине, чтобы стимулировать появление идей. В общем, как бы то ни было, вкладывайтесь в решение задачи, выделяя время на размышления о ней.

Жертвуйте точностью, если она стоит на пути к хорошей модели

Многие ошибочно думают, что предметная модель должна в точности соответствовать реальности; в действительности же моделировать следует не реальную жизнь во всей ее полноте, а скорее полезные абстракции предметной области. Ищите в предметной области общее и частное. Определите, что из этого склонно к изменениям и должно рассматриваться как сложное. Используйте эту информацию для построения своей модели. Это намного полезнее, чем вычленять из предметной области существенные и глаголы. И самое главное: включайте в модель только то, что необходимо для выполнения бизнес-сценариев использования.

Предметная модель — это не слепок реальности, а система абстракций реальности, интерпретация, которая включает в себя лишь те аспекты предметной области, которые важны для решения конкретной бизнес-задачи. В предметной модели не должно быть никаких лишних деталей предметной области, не служащих достижению этой цели. Карта лондонского метрополитена, приведенная на рис. 4.8, спроектирована для решения определенной задачи. Она не является полноценным отражением реальности: по ней невозможно определять расстояния между достопримечательностями Лондона, однако она весьма полезна для поездок на метро. Она проста и эффективна в том контексте, для которого была создана.

Поскольку предметная модель не служит отражением реального мира, ее невозможно охарактеризовать как верную или неверную. Вместо этого следует говорить о ее полезности или бесполезности с позиций той задачи, для решения которой она предназначена.

Создание эффективной предметной модели является основой DDD. Это артефакт переработки и распространения знаний, постижения предметной области

¹ CRUD — от англ. Create, Read, Update, Delete (создать, прочитать, изменить, удалить). — *Примеч. пер.*

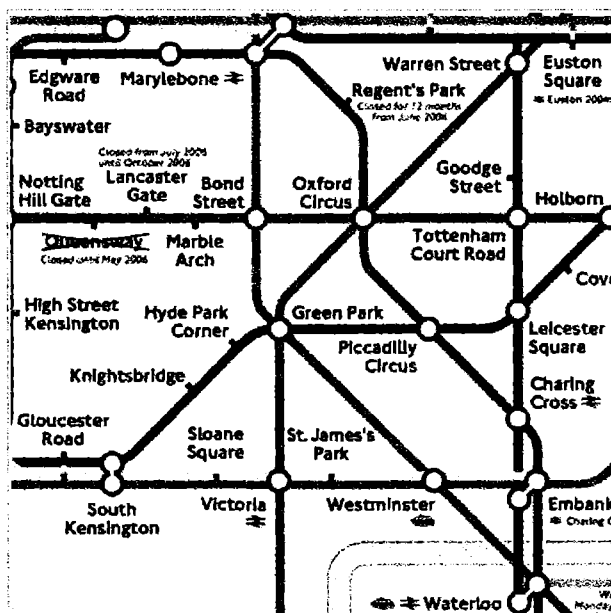


Рис. 4.8. По карте лондонского метрополитена сложно судить о расстоянии между станциями

и открытий, совершенных в ходе проектирования. Наличие полезной модели с богатым единым языком является ключом к достижению бизнес-целей предметной области. Создание практичной предметной модели — сложная задача, для решения которой требуются интенсивные исследования, эксперименты, совместная работа с экспертами и обучение.

Включайте в модель только то, что имеет отношение к задаче

Предметная модель нужна исключительно для разработки приложения. Будьте избирательны при ее создании — нет нужды включать в нее все подряд. Любой бизнес — большая и сложная структура, в которой много чего происходит. Попытка воплотить весь этот мир в единственной модели — затея в лучшем случае безрассудная, а в худшем — бессмысленная и чреватая колоссальной потерей времени, не говоря уж о том, что ее сопровождение станет сущим кошмаром. Если вы моделируете большую систему, разбейте ее на более мелкие и управляемые части, четко разграничив их.

Старайтесь не моделировать действительные взаимоотношения; вместо этого определяйте в системе ассоциации (значимые связи) в терминах инвариантов и правил. В реальной жизни у клиента обычно есть кредитная история и адрес электронной почты, но как часто вы сталкивались с требованием иметь хорошую кредитную историю и электронный адрес, начинающийся с «А», чтобы заказать какой-то товар? Группируя данные и поведение, руководствуйтесь потребностями

ми предметной области, а не своими представлениями о том, что чему должно соответствовать. Помните, что вы создаете модель, которая должна удовлетворять определенному бизнес-сценарию использования (или набору таких сценариев), а не отражать реальный мир во всей его полноте.

Чтобы ваша предметная модель оставалась конкретной и актуальной, постоянно проверяйте ее на новых сценариях и уточняйте свое понимание, обращаясь к экспертам в предметной области. Убирайте из модели любое поведение, потерявшее актуальность, чтобы избежать ее захламления.

Предметные модели полезны лишь временно

Чтобы оставаться полезной, предметная модель должна постоянно уточняться. Модель полезна лишь на протяжении некоторого интервала времени для данной итерации и данных вариантов использования. Изменения в бизнесе или появившиеся позже варианты использования могут сделать модель бесполезной. Предметная модель представляет собой реализацию общего языка, применимую только в данный отрезок времени. Разработчики должны понимать это и не привязываться к какой-то определенной модели, какой бы элегантной она ни была. Нужно быть готовым в любой момент оставить прежнюю модель и начать создавать новую, если прежняя стала неактуальной.

Используйте недвусмысленную терминологию

Способность к эффективной коммуникации — наиболее важное для решения задач качество. Цель разработчика состоит не в том, чтобы написать программный код, а в том, чтобы решить задачу. Именно поэтому так важно общаться с представителями того бизнеса, для которого вы работаете, на языке, не содержащем двусмысленностей и не требующем перевода. В отсутствие языковых барьеров эксперты в предметной области и разработчики могут свободно действовать вместе, исследовать разные варианты архитектуры модели и экспериментировать с ними, чтобы получить в результате полезную модель. Затем можно при помощи того же единого языка выполнить техническую реализацию и представить экспертам любые нюансы архитектуры без необходимости переводить с языка на язык и без смысловых потерь.

Ограничивайте свои абстракции

Вводите абстракции только из соображений общности и даже в этом случае старайтесь избегать их. Абстракции влекут за собой накладные расходы. Намного лучше явно выразить конкретную идею, чем в стремлении избежать повторов пытаться связать далекие друг от друга понятия в единый суперкласс, закладывая фундамент для последующих проблем и усложняя сопровождение кода.

Абстрактный класс или интерфейс должен представлять идею или понятие предметной области. Чрезвычайно важно ограничивать проникновение абстракций в программный код, создавая их только для тех предметных понятий, у которых есть разновидности. Не стремитесь определить абстракции для всего, что есть в предметной области. Если какое-то понятие не предполагает наличия разновидностей, используйте для его реализации конкретный программный код, а абстракт-

ции применяйте только тогда, когда нужно охватить несколько отличающихся версий понятия. Помните, что всегда лучше сохранять конкретику, чем прятать важные предметные понятия под наслоениями ненужных абстракций.

Так когда стоит прибегать к абстракциям? Возьмем в качестве примера путь на работу. Абстрактным понятием для этого случая могло бы быть «перемещение», тогда как «прогулка пешком», «поездка на поезде» или «поездка на автомобиле» выступали бы в качестве разновидностей этого понятия, то есть конкретных реализаций. Если бы способ добраться до работы был единственным (скажем, мы все ездили бы на автомобилях), нам не пришлось бы вводить такую абстракцию, как перемещение.

Применяйте абстракции на правильно выбранном уровне

Эффективная предметная модель должна отражать бизнес-сценарии использования, вводя в код абстракции на правильно выбранном уровне. Те, кому придется читать код, должны иметь возможность быстро схватить основные предметные понятия, не углубляясь в изучение программного кода.

Создавайте абстракции на самом высоком уровне; слишком большое количество абстракций на низком уровне может вызвать ненужные сложности, когда понадобится переработать модель для охвата нового сценария или новых соображений, возникших в ходе проектирования.

Абстракции всегда влекут за собой дополнительные накладные расходы, поэтому необходимо вводить их на правильном уровне и только в тех частях кода, где они принесут пользу. Избегайте применения абстракций на слишком низком уровне и вместо них используйте прием композиции поведения конкретных объектов. Абстракции образуют зависимости между классами, а рост количества зависимостей повышает связность кода.

Описывайте в абстракциях поведение, а не реализацию

Не следует определять абстракции для решения конкретных узких задач; абстракции должны представлять обобщенные понятия, такие как, например, интерфейс `IShippingNoteGenerator` в приложении обработки заказов. Разновидностями этого понятия будут «внутренняя доставка» и «международная доставка», которые отличаются составом оформляемых документов. Не следует автоматически создавать абстракции для родственных понятий. Продолжая пример с предметной областью системы исполнения заказов, не следует пытаться создать обобщенную абстракцию для каналов курьерской доставки: такая абстракция отражает инфраструктурный аспект и не служит для представления поведения предметной области. Реализация подобных аспектов должна быть конкретной, явной и находиться за пределами предметной модели. Говоря о предметных понятиях, мы в действительности говорим о поведении предметной области. Создавайте абстрактные классы и интерфейсы на основе этого поведения, следите за тем, чтобы они были небольшими и хорошо сфокусированными. Спросите себя, сколько вариантов поведения есть у данного аспекта предметной области. Не вводите абстракции насильно; используйте их только тогда, когда они помогают более ясно отразить в модели понятия предметной области.

Как и шаблоны проектирования, понятия предметной области нередко проявляются в ходе рефакторинга кода. Когда это происходит и вы обнаруживаете несколько разновидностей одного и того же понятия, тогда можно ввести абстракцию в виде интерфейса или абстрактного класса. Помните также о рисках преждевременного рефакторинга. Когда вы недостаточно хорошо знакомы с предметной областью, вы можете не увидеть наилучший путь рефакторинга. Не загоняйте себя в угол — позвольте программному коду развиваться на протяжении нескольких итераций, а затем поищите закономерности, естественным образом возникшие вокруг соответствующего аспекта поведения. Это гораздо более удачная отправная точка для рефакторинга и создания абстракций.

Окиньте взглядом все абстракции в вашей системе. Что ваши интерфейсы и базовые классы говорят о предметной области приложения? Они должны раскрывать основные понятия системы, а не просто выступать в качестве абстракций каждой реализации поведения.

Воплощайте модель в коде как можно раньше и чаще

Жизненно важно проверять архитектуру модели, перенося ее в код и тестируя на предметных сценариях: это позволяет убедиться в правильности своих умозрительных представлений, а также вскрыть любые технические ограничения, требующие изменения модели или поиска компромиссных решений. Техническая реализация поможет обнаружить любые проблемы в архитектуре модели и кристаллизует ваше понимание предметной области.

Не останавливайтесь на первой же работоспособной идее

Не хватайтесь за первую же работоспособную идею — прекращайте моделирование только тогда, когда идеи исчерпались. Получив полезную модель, возвращайтесь к началу процесса — поставьте себе целью построить модель по-другому, экспериментируйте, используя все свои знания и навыки. Попробуйте решить задачу с совершенно иной моделью. Если у вас не получится с первого раза, поищите более удачное решение. Постоянно пересматривайте свои представления о предметной области — это поможет создать более выразительную модель. С углублением знаний модель будет изменяться.

Помните, что всякая модель полезна только в данный отрезок времени; старайтесь не заикливаться на найденном решении, даже если оно выглядит изящным. Отбрасывайте те части модели, которые стали бесполезными, и будьте готовы изменять модель с появлением новых сценариев и вариантов использования.

Когда следует применять проектирование на основе модели

Простые задачи не требуют сложных решений. Не нужно создавать единый язык для всего приложения в целом. Направьте свои усилия и усилия экспертов в пред-

метной области на сложные и важные части — на смысловое ядро. Не тратьте силы попусту на неспециализированные и поддерживающие области, особенно если они не содержат логики, характерной для данной предметной области, — иначе вы утомите экспертов и у них не будет никакого желания помогать вам, когда дело дойдет до действительно сложных аспектов приложения.

Если вы столкнулись с какой-то сложной областью, испытываете трудности во взаимодействии с заинтересованными лицами или ваша команда приступила к работе над той частью предметной области, о которой вы знаете слишком мало, самое время переключиться на моделирование и работу над единым языком.

Всегда задавайте себе вопрос: «Не вышел ли я за рамки смыслового ядра? Является ли данная задача частью сложно устроенной предметной области? Интересна ли эта часть приложения для бизнеса? Существенно ли ее влияние на ситуацию? Насколько она важна для бизнеса, возлагаются ли на нее большие надежды — или бизнесу достаточно, чтобы эта часть просто работала?»

Не тратьте силы на моделирование, если это того не стоит

Если вы столкнулись с особенно сложным и неприятным пограничным случаем в той области системы, которая не является смысловым ядром, подумайте о том, чтобы реализовать его с помощью ручной процедуры. Отказ от автоматизации пограничных случаев и преобразование их в явные ручные процедуры помогает сберечь ценное время и высвобождает ресурсы для работы над смысловым ядром. Люди, работая вручную, прекрасно справляются с пограничными случаями и часто способны принимать на основе данных такие решения, для воспроизведения которых в коде потребовалось бы значительное время.

Сосредоточьтесь на смысловом ядре

Смысловое ядро приложения — это та причина, по которой приложение разрабатывается, а не приобретается на стороне. Именно эта область вызывает больше всего энтузиазма у заинтересованных лиц, именно в ней вас ждут самые интересные обсуждения, и именно с ней связаны самые ценные встречи по переработке знаний. Это та область, где единый язык приносит больше всего пользы и требует больше всего внимания. Не старайтесь создать богатый единый язык для всей предметной области, поскольку для многих неспециализированных и поддерживающих областей он попросту не нужен и станет напрасной тратой сил. Сосредоточьте свои усилия на том, что действительно ценно. Не пытайтесь охватить единым языком всё и вся. В тех аспектах приложения и предметных подобластях, для которых высокая сложность не характерна, единый язык не даст вам особых преимуществ, поэтому не разбрасывайтесь по мелочам. Смысловое ядро обычно невелико; сосредоточьтесь на нем. Создание единого языка обходится дорого.

Ключевые идеи

- Предметная область — это реальность вокруг решаемой задачи. Модель предметной области — это набор абстракций, основанных на таком представлении предметной области, которое призвано обслуживать конкретные бизнес-сценарии использования.
- Предметная модель представляется в виде аналитической и программной моделей. Они составляют единое целое.
- При проектировании на основе модели аналитическая модель увязывается с программной посредством использования общего языка.
- Аналитическая модель представляет ценность только до тех пор, пока остается согласованной с программной моделью.
- Работая над аналитической или программной моделью, вам придется приложить руку к созданию программного кода. Здесь найдется работа архитекторов, но они должны быть при этом и программистами.
- Программный код — основная форма выражения модели. Он должен быть тесно связан с моделью единым языком.
- Процесс создания единого языка является наиболее важной частью философии предметно-ориентированного проектирования, поскольку обеспечивает возможности общения и обучения.
- Предметную терминологию нужно задать явным образом, чтобы обеспечить ее смысловую точность, поскольку терминология, используемая при общении, неизбежно проникает в программную реализацию.
- Неявно присутствующие в предметной области идеи, которые должны стать понятны разработчикам, делаются явными и получают названия, образующие общий единый язык.
- Для описания сложных концепций в ясной и сжатой форме в предметных областях есть богатая специальная терминология и собственный язык.
- Понять поведение системы можно из описания ее особенностей и сценариев использования, однако эксперты в предметной области помогут вам сконструировать модель, способную поддерживать заданное поведение.
- Единый язык должен использоваться повсеместно — в тестах, пространствах имен, именах классов и методов.
- Общение — важный аспект, требующий особой заботы; единый язык создается прежде всего для совместной работы и не подразумевает, что разработчики просто примут на вооружение язык бизнеса.
- Используйте предметные сценарии, чтобы убедиться в полезности модели и проверить, насколько глубоко команда понимает предметную область.
- Применяйте проектирование на основе модели и создавайте единый язык только для работы над смысловым ядром, которое имеет особое значение для бизнеса. Не распространяйте использование этих приемов на разработку всех частей приложения.

5

Шаблоны реализации предметной модели

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Роль уровня предметной области в приложении
- Шаблоны реализации предметной модели в программном коде
- Как выбрать правильный шаблон проектирования для представления модели

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign, на вкладке **Download Code** (Загружаемый код). Примеры кода для главы 5 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Главной задачей DDD является уменьшение сложности. Как вы уже знаете, достигается это за счет помещения модели предметной области в сердце программного обеспечения и реализации на ее основе поведения приложения. Существуют разные шаблоны, помогающие представить модель в форме программного кода. В главе 3 «Концентрация на смысловом ядре» вы познакомились с подобластями и узнали, что в основе крупных приложений может лежать несколько моделей. Однако не все модели одинаково сложны или важны. Некоторые из них могут описывать сложную предметную логику, другие просто отвечают за управление данными, поэтому для каждой модели следует выбирать свой шаблон проектирования, наилучшим образом представляющий свою модель в программном коде.

Важно понимать, что не существует универсальных рекомендаций по выбору шаблона проектирования для представления предметной логики. Если предметная логика надежно изолирована от технических проблем, можно с успехом применять тактику проектирования на основе модели (Model-Driven Design) и, соответственно, шаблон предметной модели.

Эта глава представляет вашему вниманию шаблоны проектирования, которые можно использовать для реализации предметной модели. Наряду с описанием

каждого шаблона будут даны рекомендации, в каких случаях его применение целесообразно, а в каких лучше воздержаться от использования.

Уровень предметной области

Уровень предметной области, лежащий в основе приложения, — это фрагмент программного кода, описывающий предметную модель. Он отделяет сложности предметной модели от любых технических сложностей самого приложения. Он также является гарантией того, что такие проблемы инфраструктуры, как, например, управление транзакциями и поддержание текущего состояния системы, не станут проблемами бизнес-задач и не приведут к нарушению правил, действующих в предметной области. В большинстве случаев предметный уровень составляет лишь малую часть приложения. Все остальное принадлежит инфраструктурному уровню и уровню представления, как показано на рис. 5.1.

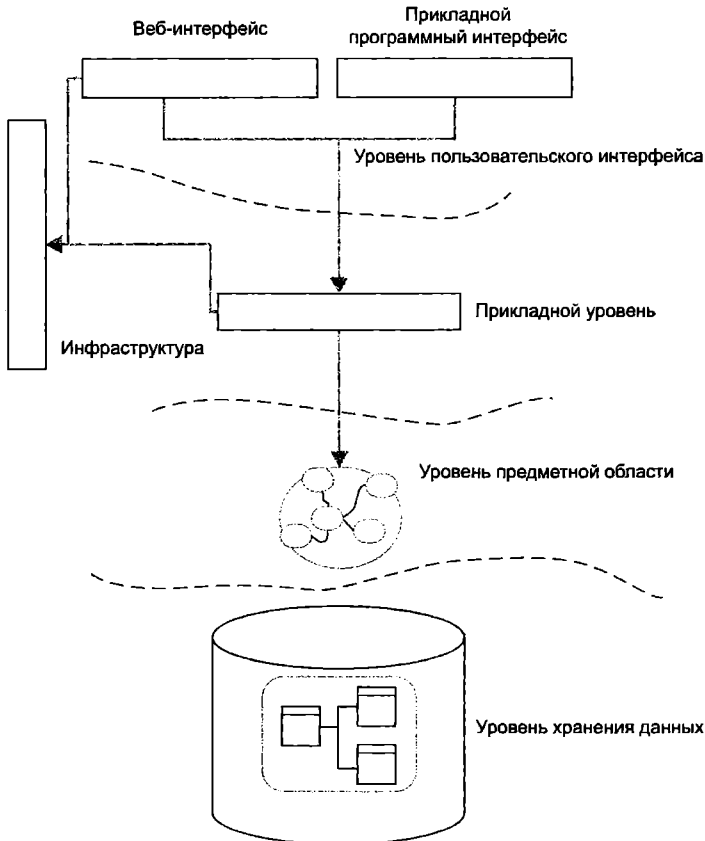


Рис. 5.1. Программный код реализации предметной модели составляет лишь малую часть приложения

Шаблоны реализации предметной модели

Существует множество различных шаблонов реализации предметных моделей в программном коде. Большие системы могут быть спроектированы по-разному. Некоторые части являются более важными, другие – менее, и разные модели используются для различных контекстов. На рис. 5.2 изображено сосуществование нескольких моделей в одном приложении. Это обусловлено тем, что для разных контекстов необходимы разные модели или к работе привлечено несколько команд разработчиков, каждая из которых работает над своей моделью. Границы моделей подробно рассматриваются в главе 6 «Обеспечение целостности моделей»

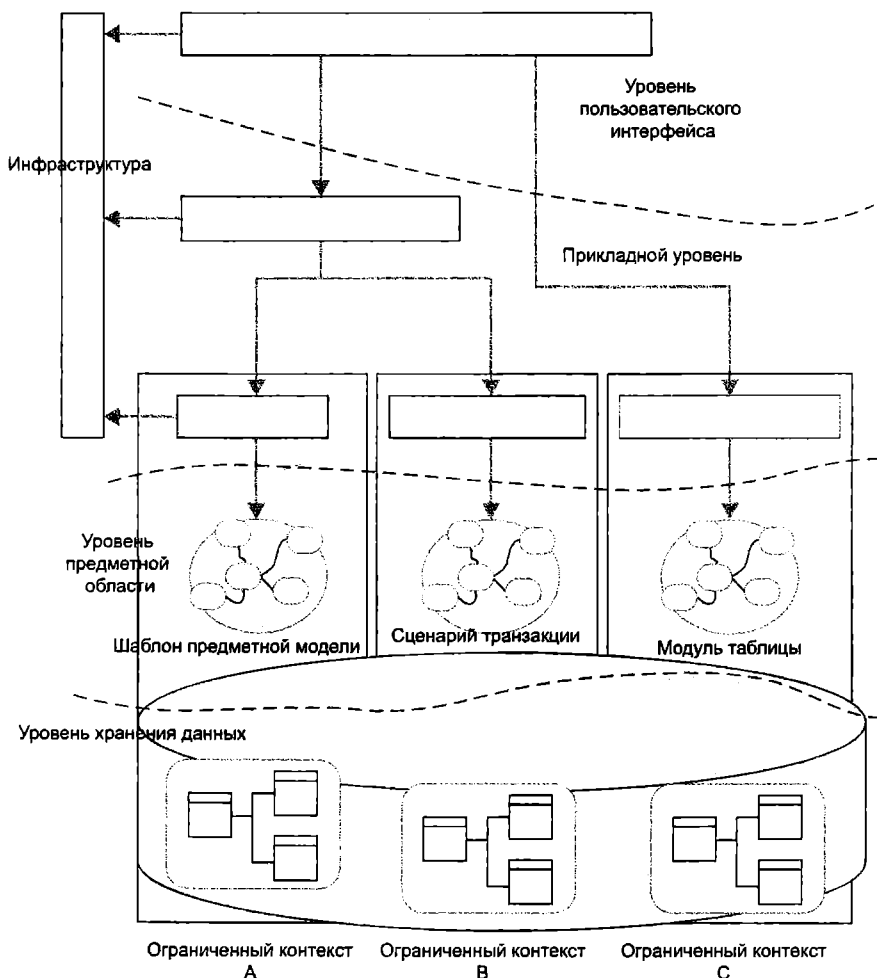


Рис. 5.2. Несколько предметных моделей, реализованных с применением разных шаблонов, в составе одного приложения

предметной области с помощью ограниченных контекстов». На данном этапе достаточно уяснить, что в процессе может быть задействовано несколько моделей и все эти модели могут быть по-разному реализованы. На рис. 5.2 показан пример, как большое приложение можно разделить на контексты с разными шаблонами представления предметных моделей. В оставшейся части главы мы займемся исследованием шаблонов проектирования, чтобы вы могли взять их за образец при моделировании предметной области. Три шаблона, описанные далее, впервые были представлены в книге «Архитектура корпоративных программных приложений» Мартина Фаулера (Martin Fowler):

- «Предметная модель»;
- «Сценарий транзакции»;
- «Модуль таблицы».

Помимо шаблонов, предложенных Мартином Фаулером, вы также познакомитесь с шаблонами «Активная запись», «Анемичная модель предметной области» и несколькими функциональными шаблонами. Выбор того или иного шаблона из числа описываемых в этой главе зависит от сложности каждой модели в приложении.

Предметная модель

Шаблон «Предметная модель» (Domain model), предложенный в книге Мартина Фаулера «Архитектура корпоративных программных приложений», синонимичен понятию DDD, потому что хорошо подходит для сложных предметных областей с разветвленной бизнес-логикой. Шаблон «Предметная модель» — это объектно-ориентированная модель, описывающая как поведение, так и данные. На первый взгляд он может показаться аналогичным модели хранения данных (схеме базы данных в терминологии реляционных БД). Однако несмотря на то что и там и там подразумевается хранение данных, шаблон «Предметная модель» включает также бизнес-процессы и связи, правила и разветвленную предметную логику. Методология DDD предлагает множество простых функциональных блоков, описываемых в третьей части книги, которые позволяют эффективно реализовать шаблон Фаулера «Предметная модель».

Шаблон «Предметная модель» опирается на предположение об отсутствии базы данных; следовательно, в процессе его создания идеей о необходимости сохранения данных можно полностью пренебречь. Проектируя модель, вы начинаете не с организации данных, а с программной модели — проектирование на основе модели (model-driven design) как бы противопоставляется проектированию на основе данных (data-driven design). Только в случае возникновения потребности в сохранении данных можно пойти на компромисс с проектированием. Предметные объекты в модели квалифицируются как простые классы объектов C# (Plain Old C# Objects, POCO). Для этих классов характерно отсутствие зависимости от инфраструктурных проблем и пренебрежение способами хранения и организации

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: ООО «И. Д. Вильямс», 2010. (Ранее выходила в том же издательстве под названием «Архитектура корпоративных программных приложений».) — *Примеч. пер.*

данных. На рис. 5.3 показано, как разделены шаблон «Предметная модель» и техническая инфраструктура.

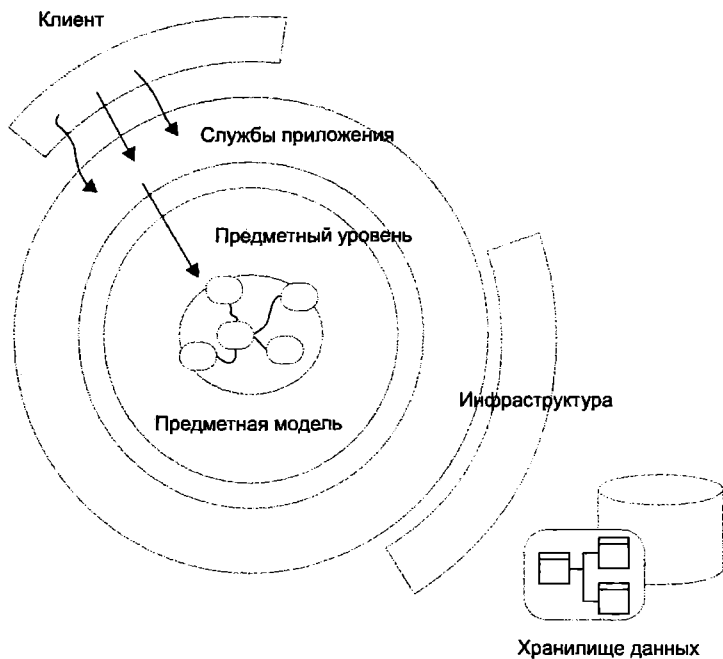


Рис. 5.3. Шаблон «Предметная модель»

Такая возможность сосредоточиться только на предметной модели позволяет проектировать предметную логику на основе абстракций предметной области, что отвечает философии DDD. Отсутствие необходимости думать о сохранении данных позволит создать выразительную модель, заточенную непосредственно под проблематику данной предметной области. Конечно, вам понадобится сохранять какие-то данные и ради этого придется позже искать компромиссное решение, но задумываться об этом на этапе моделирования не нужно. Благодаря этому модель не будет отягощена инфраструктурным кодом, ее направленность будет касаться исключительно предметной логики.

Предметная модель может представляться как некий концептуальный уровень, отражающий предметную область, над которой вы работаете в данный момент. В этой модели присутствуют различные объекты предметной области и связи между ними. Например, вы занимаетесь разработкой приложения электронной коммерции, тогда «объектами» такой модели будут корзина покупателя, заказ, детали заказа и т. п. Все эти объекты содержат какие-то данные и, что самое важное, обладают некоторым поведением. Так, объект «заказ» не только имеет характеристики, отражающие дату его оформления, состояние и номер, но и может быть дополнен бизнес-логикой. Сюда относится, например, применение скидок по купону, а также все связанные с этим правила предметной области: действителен ли

купон, применим ли он к товарам в корзине, нет ли каких-либо других условий, которые повлекут невозможность использовать купон.

На рис. 5.4 показана часть предметной модели сайта электронных торгов. Объекты в модели представляют собой понятия предметной области, которые используются для описания поведения аукциона. Как можно заметить, модель построена на терминах аукционной деятельности, выраженных именами существительными, однако это не всегда бывает так. На самом деле при работе над моделью вам лучше брать во внимание глаголы и действия предметной области, это поможет сконцентрироваться на поведении, а не на состоянии и в конечном результате получить объектно-ориентированное представление модели данных.

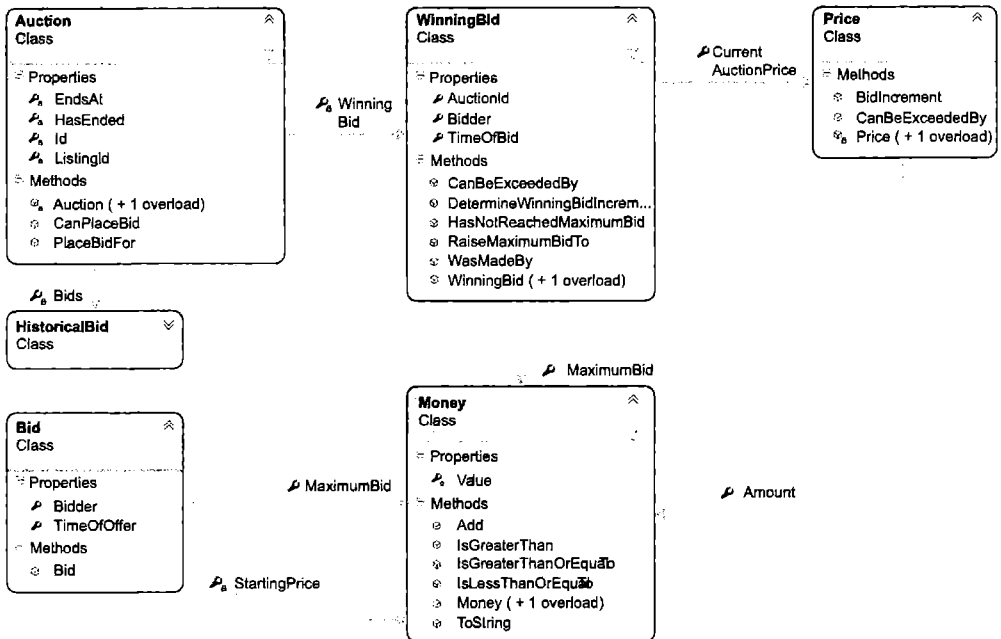


Рис. 5.4. Предметная модель сайта электронных торгов

Каждый объект в предметной модели отвечает за решение определенной задачи. Объекты действуют в связке, следуя бизнес-сценариям, делегируя задачи друг другу. В листинге 5.1 можно увидеть, как класс `Auction` обращается к `WinningBid`, чтобы получить новое значение цены лота после повышения ставки.

Листинг 5.1. Класс `Auction` из предметной модели с разветвленной логикой

```

public class Auction
{
    ...
    public void PlaceBidFor(Bid bid, DateTime currentTime)
    {

```

```
if (StillInProgress(currentTime))
{
    if (FirstOffer())
        PlaceABidForTheFirst(bid);
    else if (BidderIsIncreasingMaximumBid(bid))
        WinningBid = WinningBid.RaiseMaximumBidTo(bid.MaximumBid);
    else if (WinningBid.CanBeExceededBy(bid.MaximumBid))
    {
        Place(WinningBid.DetermineWinningBidIncrement(bid));
    }
}
}
...
}
```

Шаблон «Предметная модель» прекрасно подходит для реализации решений в сложных предметных областях. Именно такой исключительный объектно-ориентированный подход позволяет создать абстрактную модель реальной области бизнеса и является удобным при работе со сложной логикой и производственными процессами. Предметная модель полностью игнорирует проблемы, связанные с сохранением данных, опираясь в этом на классы отображения элементов и другие абстракции, обеспечивающие сохранение и получение объектов. Если вам необходимо описать в модели сложную логику или часть предметной области, где особо важна ясность или предполагаются частые изменения ввиду поступления дополнительных инвестиций, то лучшего решения, чем данный шаблон, не найти.

Однако шаблон «Предметная модель» нельзя назвать универсальным решением, его использование может оказаться затратным. Он считается одним из самых сложных в плане технической реализации и требует привлечения разработчиков с серьезным бэкграундом в области объектно-ориентированного программирования. Большинство подсистем опирается на простые CRUD-операции, в которых шаблон «Предметная модель» может потребоваться только при реализации смыслового ядра для обеспечения ясности или управляемости сложной логики. Поэтому вам точно не следует пытаться применять этот шаблон везде. Некоторые части вашего приложения будут представлять собой простые формы с данными, требующими лишь обычной проверки и не подразумевающими сложной бизнес-логики. Попытка применить приемы объектно-ориентированного проектирования для моделирования всего подряд может стать напрасной тратой сил, которые лучше направить на реализацию смыслового ядра. Разработка программного обеспечения — это история об упрощении вещей, так что если вы имеете дело со сложной логикой, применяйте шаблон «Предметная модель»; в противном случае выберите другой шаблон, который лучше подходит для решения стоящей перед вами задачи, например «Анемичная предметная модель» или «Модуль таблицы».

Если часть приложения, над которой вы работаете, не имеет часто изменяющейся логики и представляет собой простую форму с данными, для ее реализации лучше не использовать шаблон «Предметная модель». Использование этого шаблона не по назначению в лучшем случае приведет к напрасной трате сил, а в худшем —

к ненужному усложнению там, где было бы достаточно применить более простой метод проектирования.

Сценарий транзакции

Из всех шаблонов проектирования предметной логики, с которыми вы встретитесь в этой главе, шаблон «Сценарий транзакции» (Transaction script) является самым простым для понимания и применения. Этот шаблон больше соотносится с процедурным подходом к программированию, нежели с объектно-ориентированным. При таком подходе для каждой бизнес-транзакции в вашем проекте создаются отдельные процедуры, которые затем группируются в некоторый статический управляющий или служебный класс. Каждая процедура содержит всю необходимую бизнес-логику для успешного выполнения бизнес-транзакции, начиная от производственных процессов, правил и валидации данных до сохранения информации в базе данных.

На рис. 5.5 изображено графическое представление шаблона «Сценарий транзакции».

На рис. 5.6 показан пример сигнатуры интерфейса, реализующего шаблон «Сценарий транзакции». Две реализации интерфейса содержат всю логику, необходимую для создания аукциона (слева) и ведения ставок на аукционе (справа), включая логику доступа к данным в хранилище, авторизацию, координацию параллельных транзакций и обеспечение стабильности системы.

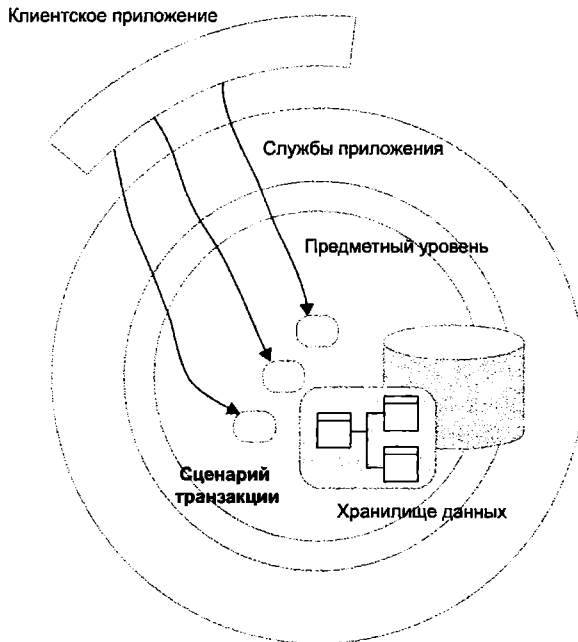


Рис. 5.5. Шаблон «Сценарий транзакции»

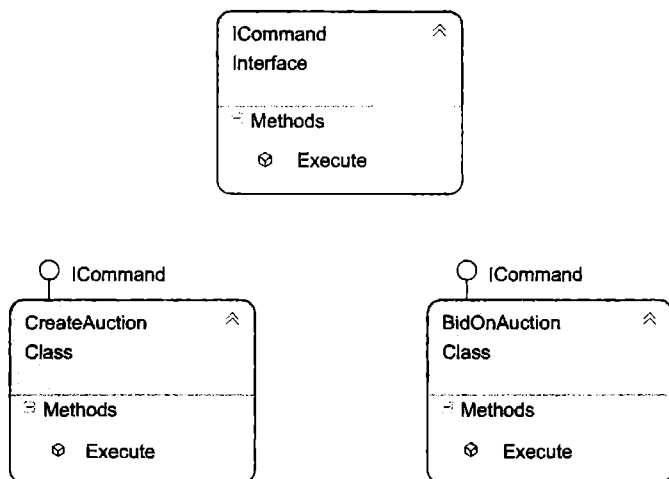


Рис. 5.6. Схема UML шаблона «Сценарий транзакции»

Одними из плюсов применения шаблона «Сценарий транзакции» являются его простота и скорость, с какой могут включаться в работу новые разработчики, не имеющие опыта его использования ранее. С введением дополнительных требований к разработке приложения вы легко сможете добавлять в класс новые методы, не опасаясь нарушить работу существующих.

Продолжая разбирать пример сайта аукциона, рассмотрим листинг 5.2, где демонстрируется реализация выполнения ставки на аукционе с применением шаблона «Сценарий транзакции».

Листинг 5.2. Сценарий использования, смоделированный с применением шаблона «Сценарий транзакции»

```

public class BidOnAuction: ICommand
{
    public BidOnAuction(Guid auctionId, Guid bidderId,
        decimal amount, DateTime timeOfBid)
    {
        // ...
    }

    public void Execute()
    {
        using (TransactionScope scope = new TransactionScope())
        {
            ThrowExceptionIfNotValid(auctionId, bidderId, amount, timeOfBid);

            ThrowExceptionIfAuctionHasEnded(auctionId);

            if (IsFirstBid(auctionId))
                PlaceFirstBid(auctionId, bidderId, amount, timeOfBid);
            else if (IsIncreasingMaximimBid(auctionId, amount, bidderId))

```

```
        IncreaseMaximumBidTo(amount);
    else if (CanMeetOrExceedBidIncrement(amount))
        UpdatePrice(auctionId, bidderId, amount, timeOfBid);
    }
}
...
}
```

Как видите, весь сценарий использования инкапсулирован в единственный метод. Класс отвечает за выполнение множества операций, включая извлечение и сохранение данных, управление транзакциями, а также за реализацию бизнес-логики по размещению ставки на аукционе.

Шаблон «Сценарий транзакции» — простой процедурный шаблон, удобный для реализации разделов предметной области, не обладающих сложной логикой. Вся логика для одной операции заключена в единственный метод. Любой разработчик с легкостью освоит архитектуру, используемую для моделирования предметной логики. Именно поэтому данный шаблон удобен командам, в состав которых входят начинающие разработчики, пока не очень уверенно оперирующие концепциями объектно-ориентированного программирования. Однако как только дело доходит до усложнения предметной логики, шаблон «Сценарий транзакции» быстро становится неуправляемым из-за свойственного ему дублирования программного кода. Так что если вы обнаружили в реализации чрезмерное дублирование кода, следует произвести его рефакторинг в сторону шаблона «Предметная модель».

Недостатки шаблона «Сценарий транзакции» начинают проявляться с ростом приложения и усложнением бизнес-логики. Расширение приложения повлечет увеличение числа методов, часто малополезных и узкоспециализированных, заполняющих API приложения и перекрывающих функциональность друг друга. Вам может прийти мысль использовать вспомогательные методы, чтобы избежать дублирования программного кода, например, при проведении проверки данных и соответствия бизнес-правилам, но избавиться от дублирования в реализациях рабочих процессов, увы, не удастся, поэтому с ростом приложения программный код быстро может стать громоздким и неуправляемым.

Модуль таблицы

Шаблон «Модуль таблицы» (Table module) отображает объектную модель в виде модели базы данных. Каждому объекту соответствует таблица или представление (view) в базе данных. Таблицы позволяют сохранять данные и реализовывать бизнес-логику. Преимущество этого шаблона в том, что между объектной моделью и моделью базы данных нет рассогласований. Шаблон «Модуль таблицы» отлично подходит для проектирования на основе данных (Database-Driven Design), поэтому на первый взгляд может показаться, что он плохо согласуется с философией DDD. Однако для реализации простых разделов предметной области, изолированных посредством ограниченных контекстов и представляющих собой простые формы с данными, этот шаблон подходит как нельзя лучше и осваивается разработчиками

намного быстрее, чем шаблон «Предметная модель». Тем не менее, если вы выявили расхождения между объектной моделью и моделью базы данных, необходимо прибегнуть к рефакторингу кода в сторону шаблона «Предметная модель».

Активная запись

Шаблон «Активная запись» (Active record) является разновидностью модуля таблицы и отображает объекты в виде ее строк, а не в форме самостоятельных таблиц, как в предыдущем шаблоне. Объект представляет собой строку (запись) базы данных в переходном состоянии или во время редактирования.

Шаблон «Активная запись» довольно популярен благодаря своей эффективности в тех случаях, когда основополагающая модель базы данных соответствует предметной модели. Как правило, для каждой таблицы в базе данных существует свой бизнес-объект, представляющий собой отдельную строку этой таблицы. В строке содержатся данные и описание определенного поведения, а также способы сохранения этих данных и методы для добавления новых сущностей и поиска библиотек объектов. В шаблоне «Активная запись» каждый бизнес-объект сам отвечает за свое сохранение и связанную бизнес-логику.

Шаблон «Активная запись» отлично подходит для простых приложений, когда имеется прямое соответствие между моделью данных и предметной моделью. Примерами могут служить движки для блогов и форумов. Этот шаблон также с успехом можно использовать, если модель базы данных уже существует или приложение конструируется по принципу «сначала данные». Так как каждый бизнес-объект отображается в виде отдельной таблицы базы данных и все они имеют общие методы создания, чтения, обновления и удаления (create, read, update и delete — CRUD), то становится возможным воспользоваться инструментами автоматической генерации программного кода для созданной вами предметной модели. Хорошие инструменты такого класса способны также простроить в базе данных всю логику валидации, чтобы гарантированно записывать только допустимые данные.

Анемичная предметная модель

Шаблон «Анемичная предметная модель» (Anemic domain model) иногда называют антишаблоном. На первый взгляд он очень похож на шаблон «Предметная модель» в том смысле, что вы снова столкнетесь с объектами, определяющими предметную область. Однако в этом шаблоне отсутствует описание поведения предметных объектов, оно определяется за пределами модели. Предметные объекты остаются лишь как классы носителей информации. Главный недостаток этого шаблона в том, что программный код реализуется более в процедурном стиле, подобно шаблону «Сценарий транзакции», с которым вы познакомились выше, что влечет те же проблемы, свойственные этому шаблону. Одной из таких проблем является нарушение принципа «говори, а не спрашивай» («Tell, Don't Ask»), который подразумевает, что объекты должны сообщать клиентскому приложению, что они могут или не могут сделать, вместо того чтобы предоставлять доступ к их свойствам и вынуждать приложение самостоятельно определять, готов ли объект для выполнения требуемой

операции. Предметные объекты в этом шаблоне почти полностью лишены какой-либо логики и являются простыми контейнерами данных.

Шаблон «Анемичная предметная модель» является хорошим кандидатом на реализацию частей предметной модели, не имеющих или имеющих незначительный объем логики, и также подходят для разработчиков без достаточного опыта объектно-ориентированного программирования. Шаблон «Анемичная предметная модель» создает предпосылки для использования единого языка, а также может стать первым шагом на пути к созданию предметной модели с разветвленной логикой.

Анемичная предметная модель и функциональное программирование

Философия предметно-ориентированного проектирования более понятна разработчикам, отдающим предпочтение функциональному стилю программирования. Предметные модели с легкостью можно строить с использованием функциональных понятий, например неизменяемости (immutability) и ссылочной прозрачности (referential transparency). Объекты с развитой логикой не являются обязательным условием успеха, так же как не является обязательной и изоляция состояний за поведенческими интерфейсами. Соответственно при использовании функционального стиля программирования шаблон «Анемичная предметная модель» превращается из антишаблона в весьма полезный инструмент.

Может показаться противоречивым, что, несмотря на тот факт, согласно которому предметные модели призваны облегчить диалог со специалистами в предметной области, шаблон «Анемичная предметная модель» исключает возможность представления предметных понятий в виде объектов. Однако, по утверждению многих практиков DDD, наиболее важными предметными понятиями являются действия, например «перевод денежных средств», а не существительные, такие как «банковский счет». Применяя шаблон «Анемичная предметная модель» и функциональное программирование, вы тем не менее можете выражать глаголами или отглагольными существительными понятия предметной области и, соответственно, вести содержательный диалог со специалистами.

В процессе конструирования функциональных предметных моделей все еще возможно использовать структуры, представляющие предметные понятия, даже при использовании шаблона «Анемичная предметная модель». Следует отметить, однако, что это самые обычные структуры данных, лишенные какого-либо поведения. Так, например, объектно-ориентированная сущность `BankAccount` с развитой логикой, представленная в листинге 5.3:

Листинг 5.3. Объектно-ориентированная сущность `BankAccount` с развитой логикой

```
public class BankAccount
{
    ...

    public Guid Id { get; private set; }
```

```
public Money Balance { get; private set; }

public Money OverdraftLimit { get; private set; }

public void Withdraw(Money amount)
{
    ...
}

public void Deposit(Money amount)
{
    ...
}

public void IncreaseOverdraft(Money amount)
{
    ...
}
}
```

может быть смоделирована в виде простой, неизменяемой структуры данных, как показано в листинге 5.4:

Листинг 5.4. Объект `BankAccount`, лишенный поведения и служащий для передачи данных

```
public class BankAccount
{
    public Guid Id { get; private set; }

    public Money Balance { get; private set; }

    public Money OverdraftLimit { get; private set; }
}
```

Адаптировав объекты в простые структуры данных, мы оставляем их поведение в виде чистых функций¹, и основная сложность заключается в том, чтобы сгруппировать их и выстроить взаимосвязь с концептуальной предметной моделью. Один из эффективных приемов решения этой задачи заключается в объединении функций в агрегаты. Другое большое отличие функционального подхода заключается в структуре функций и их применении. Так как функциональное программирование требует соблюдения принципа неизменяемости, функции должны возвращать обновленные структуры данных с необходимыми изменениями, а не изменять состояние существующих объектов. Например, при объектно-ориентированном подходе сущность `ShoppingBasket` может напрямую изменять коллекцию объектов `Items`, когда покупатель добавляет новый товар в корзину, как показано в листинге 5.5.

¹ Под чистыми функциями подразумеваются функции, всегда возвращающие один и тот же результат для одних и тех же значений аргументов и не имеющие побочных эффектов. — *Примеч. пер.*

Листинг 5.5. Объектно-ориентированный класс `ShoppingBasket`

```
public class ShoppingBasket
{
    ...

    public Guid Id { get; private set; }

    // инкапсулированное изменяемое состояние
    private List<BasketItem> Items { get; set; }

    public void Add(BasketItem item)
    {
        if (this.Items.Contains(item))
            throw new DuplicateItemSelected();
        else
            this.Items.Add(item); // изменяет состояние
    }

    ...
}
```

При использовании функционального подхода вместо изменения коллекции объектов `Items` возвращается копия `ShoppingBasket`, содержащая обновленную, неизменяемую коллекцию объектов `Items`, как показано в листинге 5.6.

Листинг 5.6. Функциональный класс `ShoppingBasket`

```
// неизменяемая структура данных
public struct ShoppingBasket
{
    public ShoppingBasket(Guid id, ImmutableList<BasketItem> items)
    {
        this.Id = id;
        this.Items = items;
    }

    public Guid Id { get; private set; }

    public ImmutableList<BasketItem> Items { get; private set; }
}

// все функции для агрегата корзины
public static class Basket
{
    // чистая функция, не принадлежит никакому экземпляру класса
    public static ShoppingBasket AddItem(
        BasketItem item, ShoppingBasket basket)
    {
        if (basket.Items.Contains(item))
            throw new DuplicateItemSelected();

        // создает новую неизменяемую коллекцию объектов
        ImmutableList<BasketItem> items = basket.Items.Add(item);
    }
}
```

```
// создает новую неизменяемую корзину
return new ShoppingBasket(basket.Id, items);
}
}
```

Обратите внимание, что коллекция объектов `Items` больше не может инкапсулироваться в класс, потому что функции не имеют доступа к приватному состоянию `ShoppingBasket`.

ВНИМАНИЕ

Будьте осторожны при создании копий объектов и коллекций. В некоторых языках программирования возвращается новая ссылка на существующий объект (такой подход называют поверхностным копированием (*shallow copy*)). Так, при копировании `ShoppingBasket` в примере с объектно-ориентированным подходом оба объекта, исходный и копия, могут ссылаться на одну и ту же коллекцию объектов `Items`. Соответственно, изменяя копию, вы измените оригинал. Чтобы избавиться от этой проблемы, убедитесь, что создаете и используете только неизменяемые структуры и коллекции объектов.

Большинство современных языков имеет встроенную поддержку неизменяемых коллекций. В C#, например, имеются неизменяемые эквиваленты для большинства изменяемых коллекций (<https://msdn.microsoft.com/ru-ru/en-us/library/dn385366%28v-vs.110%29.aspx>), а в Scala есть два модуля — `collections.mutable` и `collections.immutable`.

ПРИМЕЧАНИЕ

Сущности и агрегаты — это тактические строительные блоки, используемые практиками DDD для представления предметных понятий, таких как банковский счет, корзина покупателя или дата нахождения онлайн. Более подробно о строительных блоках DDD рассказывается в третьей части этой книги.

Некоторые языки программирования, включая Haskell, Scala и Clojure, ставят функциональное программирование в приоритет. Однако функциональные предметные модели возможно создавать и на традиционных, объектно-ориентированных языках, таких как C# и Java.

ПРИМЕЧАНИЕ

Если вы не знакомы с функциональным программированием, вам определенно стоит выделить время на изучение хотя бы самых основных понятий. Лучшим введением в функциональное программирование для новичков по общему признанию считается вики-страница проекта Haskell (https://www.haskell.org/haskellwiki/Functional_programming), даже если предполагается использовать язык программирования, отличный от Haskell.¹

¹ Замечательный учебник по функциональному программированию на русском языке можно найти по адресу https://ru.wikibooks.org/wiki/Основы_функционального_программирования. — Примеч. пер.

Ключевые идеи

- Уровень предметной модели содержит модель предметной области и изолирован от инфраструктурных задач и задач представления.
- Предметную модель можно реализовать с применением множества шаблонов предметной логики.
- В процессе проектирования большого проекта может быть задействована не одна модель и, соответственно, для реализации логики такого проекта может использоваться несколько шаблонов.
- Если шаблон изолирует программный код, описывающий предметную логику, от технического кода, такой шаблон хорошо согласуется с философией DDD.
- Шаблон «Предметная модель» хорошо подходит для сложных предметных областей. Понятия предметной области инкапсулируются в объекты, содержащих и данные, и описание поведения.
- Шаблон «Сценарий транзакции» организует всю предметную логику для реализации бизнес-процессов в процедурные модули.
- Шаблон «Модуль таблицы» представляет модель данных в форме объекта. Этот шаблон хорошо подходит для моделей, сконструированных на основе данных и наиболее точно отражающих схемы данных.
- Шаблон «Активная запись» похож на шаблон «Модуль таблицы» в том, что ставит на первое место данные, но объекты представляют собой строки в таблицах, а не сами таблицы. Такой шаблон хорошо подходит для моделей с простой логикой, широко использующих CRUD-операции.
- Шаблон «Анемичная предметная модель» напоминает шаблон «Предметная модель»; но в этом шаблоне модель полностью лишена поведения. Она моделирует исключительно состояние объекта, а вся логика размещается в обслуживающих классах.
- Функциональное программирование с равным успехом можно использовать для реализации предметных моделей.
- При использовании функционального стиля программирования описания поведения группируются в агрегаты (представляющие предметные понятия) и применяются к функциональным, неизменяемым структурам данных (также представляющим предметные понятия).

6

Обеспечение целостности моделей предметной области с помощью ограниченных контекстов

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- О проблемах архитектуры отдельных моделей
- Почему важно использовать ограниченные контексты
- Как выделять и определять границы ответственности в коде
- Как защищать целостность смыслового ядра предметной области
- Когда необходимо определять границы

В больших и сложных приложениях могут использоваться несколько моделей. Каждая такая модель будет проектироваться для описания отдельной части предметной области, и для каждой ее реализации будет выбран соответствующий шаблон кода, подходящий для уровня сложности задачи. В идеале для всех разделов предметной области должны создаваться отдельные модели; однако это возможно не всегда, так как для некоторых сложных разделов может понадобиться определить несколько моделей, а некоторые модели могут охватывать два или более раздела. Независимо от того, сколько моделей придется создать, все они должны взаимодействовать друг с другом и обеспечить полноценное функционирование системы. Именно в тех случаях, когда разработчики объединяют модели без ясного понимания контекстов, в которых те должны применяться, увеличивается вероятность потери ясности из-за смешения понятий и логики, которые эти модели представляют.

Поэтому так важно защитить целостность каждой модели и четко обозначить границы их ответственности в коде. Это достигается путем связывания моделей с определенными контекстами, известными как ограниченные контексты (bounded context). Ограниченный контекст определяется исходя из языка команды разработчиков и физических артефактов. Ограниченные контексты позволяют моделям оставаться целостными и осмысленными, что обеспечивает управляемость сложности в пространстве решений. Неукоснительное использование ограниченных контекстов играет важную роль в достижении успеха при применении предметно-ориентированного проектирования.

Проблемы архитектур с единственной моделью

Основой DDD является необходимость создания ясных и допускающих возможность дальнейшего развития программных моделей, соответствующих концептуальным моделям. Внедрение новых знаний о предметной области в такие модели может осуществляться достаточно эффективно. Однако если вся система описывается единственной моделью, понятия из одной области модели могут ошибочно связываться с внешне похожими понятиями из другой области и даже подменять их. Поэтому философия DDD настоятельно рекомендует разбивать большие и сложные системы на множество программных моделей.

Сложность модели может увеличиваться

Большие модели должны вмещать множество предметных понятий и поддерживать множество сценариев использования. Как следствие, при реализации легко ошибиться и объединить несовместимые понятия. Поиск чего-либо в большой модели также становится затруднительным. Чем больше увеличивается система, тем серьезнее становятся проблемы, препятствуя добавлению новых особенностей и усовершенствований.

Как подчеркивает рис. 6.1, с каждым новым сценарием использования и с внедрением новых знаний в модель увеличивается число понятий и зависимостей в ней, что увеличивает ее сложность.

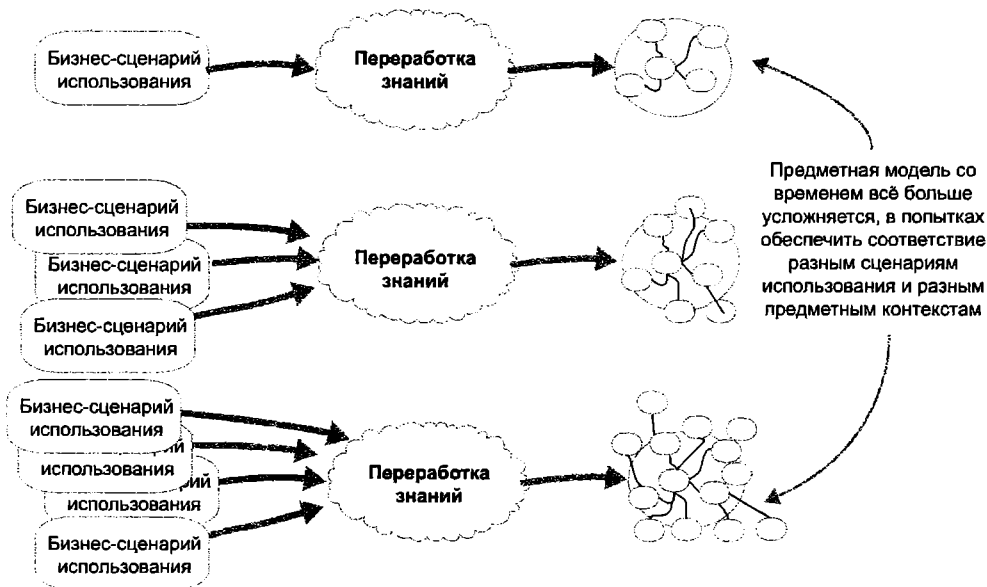


Рис. 6.1. Возрастающая сложность модели

Работа нескольких групп над единственной моделью

Сложность программного кода — лишь одна из проблем, которые влечет за собой архитектура, основанная на единственной модели. Другими важными проблемами, часто сопутствующими монолитным моделям, являются задержки, вызванные необходимостью взаимодействий между разными группами разработчиков, и организационная неэффективность.

Когда одна группа разработчиков готова выпустить обновленную версию, она должна проконсультироваться с другими группами, чтобы включить в выпуск и их изменения. В результате первая группа вынуждена ждать некоторое время или использовать сложные стратегии ветвления. Как описывается в главе 11 «Введение в интеграцию ограниченных контекстов», сложные стратегии ветвления могут быть серьезной помехой для регулярного и эффективного донесения ценности бизнеса и получения обратной связи от потребителей.

Необходимость постоянного согласования группами проектов новых особенностей дизайна или планов выпуска новых версий порождает неэффективность. Так как каждая группа работает со своим специалистом в предметной области и пытается управлять развитием модели в своем направлении, необходимость согласования планов и действий с другими группами ведет к напрасной трате сил и времени. Чем больше групп привлечено к работе над единственной моделью, тем дороже обходится организация взаимодействий между ними и тем сложнее получается программная реализация, как показано на рис. 6.2.

С другой стороны, если разбить систему на несколько моделей, группы смогут эффективно вести разработку и часто вносить новые ценные особенности, потому что они не будут вынуждены синхронизировать свои действия с другими группами или интересоваться понятиями, над которыми работают другие группы.

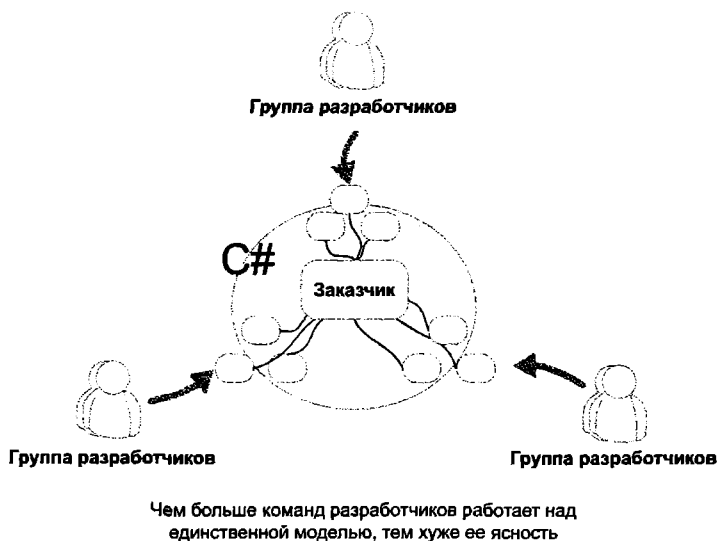


Рис. 6.2. Сложность модели возрастает с увеличением числа групп разработчиков

Кого-то может беспокоить проблема дублирования программного кода в разных моделях. Но представьте, какие выгоды несет устранение зависимостей между группами. Вообще говоря, дублирование кода в разных моделях не несет вреда, потому что в его основе лежат разные понятия.

Неоднозначность языка модели

Практики DDD часто отмечают, что некоторые понятия в системах оказываются очень похожими — они могут даже называться одинаково. Но в разных бизнес-процессах они могут иметь разный смысл. Как показано на рис. 6.3, понятие «заявка» имеет разный смысл для отдела продаж и отдела по работе с претензиями покупателей.

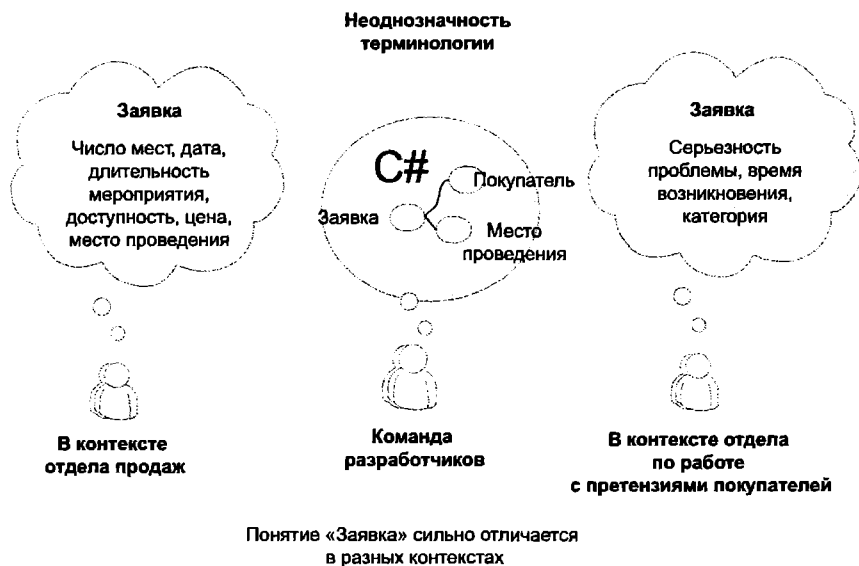


Рис. 6.3. В разных контекстах предметные термины могут иметь разный смысл

Поняв, что в разных контекстах одинаковые названия могут иметь разный смысл, вы осознали, что наличие множества мелких моделей обеспечивает более высокую эффективность, нежели одна большая модель. Кроме того, сотрудничество со специалистами также станет намного содержательнее. Например, исходя из рис. 6.3, в разговоре о заявках с руководителем отдела продаж вы понимаете, что его интересует цена и место проведения мероприятия, тогда как руководителя отдела по работе с претензиями интересуют серьезность и вид проблем, возникших у клиента.

Применимость предметных понятий

Иногда единственная физическая сущность в предметной области может ошибочно классифицироваться в программном коде как единственное понятие. Это

может вызвать проблемы, если физическая сущность в действительности представляет несколько понятий, каждое из которых по-разному интерпретируется в разных контекстах. Классическим примером является «продукт».

На рис. 6.4 показано, как сущность «продукт» может иметь разный смысл в разных контекстах. Для отдела закупок это нечто, что должно приобретаться с доходной маржой (profitable margin) и в приемлемые сроки. Однако для отдела продаж «продукт» — это понятие, связанное с фотографиями, описанием размеров и принадлежностью к категории продаж, — ни одна из этих характеристик не совпадает с характеристиками отдела закупок, даже при том, что это одна и та же физическая сущность в предметной области.

ПРИМЕЧАНИЕ

Во второй части вы узнаете, как использовать идентификаторы корреляции (correlation ID) для интеграции данных жизненного цикла физической сущности, существующей в нескольких контекстах (такой, как «продукт» в примере выше).

Когда одна физическая сущность, такая как «продукт», в действительности представляет несколько предметных понятий, она часто моделируется разработчиками как единственное понятие. К сожалению, легко попасть в ловушку, думая, что раз продукт является физической сущностью, то в программе он должен быть представлен единственным классом. Это приводит к образованию совпадений, так как каждая модель использует один и тот же класс продукта, как показано на рис. 6.5.

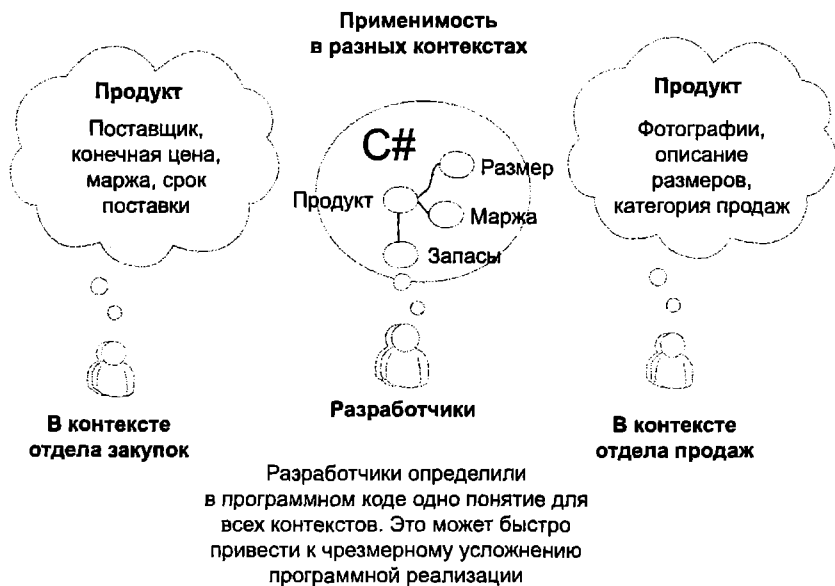


Рис. 6.4. Одно и то же понятие может иметь разную интерпретацию в разных контекстах

Приложение электронной коммерции

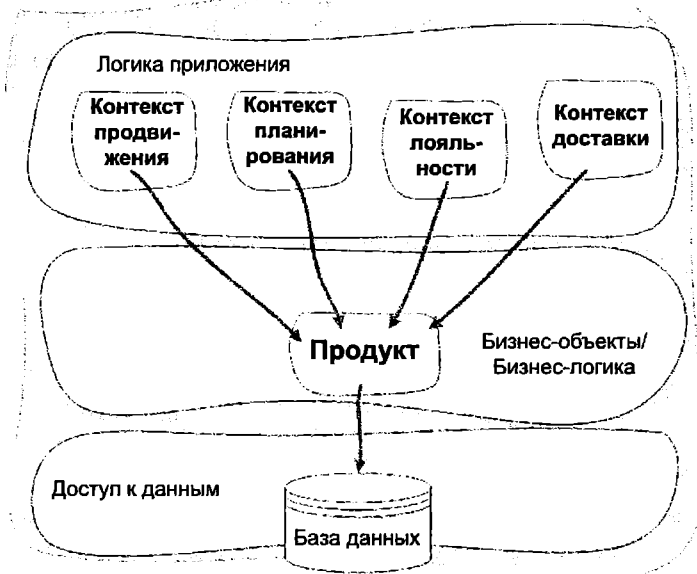


Рис. 6.5. Единое представление предметной сущности во всех контекстах может быстро привести к проблемам

Как обсуждалось выше, когда возникают совпадения между разными контекстами, программный код становится излишне сложным и возникает необходимость согласовывать действия групп разработчиков. Общий класс, в данном примере «продукт», также нарушает принцип единственной ответственности (Single Responsibility Principle, SRP), поскольку в данном случае имеется четыре контекста, в каждом из которых модель продукта должна развиваться в своем направлении.

Когда в программном коде отсутствуют четко очерченные границы, очень легко возникают совпадения, как в случае с продуктом на рис. 6.5. Лучшее решение, уменьшающее количество совпадений, это создать для каждого контекста — продвижения, закупок, лояльности и доставки — отдельную модель. В этом случае каждая модель могла бы содержать уникальное представление продукта, удовлетворяющее только потребностям контекста модели. На рис. 6.6 продемонстрировано, как выглядят множественные обязанности общего класса Product, и указывается, какие обязанности и каким моделям в действительности принадлежат.

ПРИМЕЧАНИЕ

Класс Product на рис. 6.6 — отличный пример шаблона BBoM, который мы обсуждали в первых главах. Изменение логики для одной области окажет нежелательное влияние на другие области из-за переплетения их реализаций и отсутствия четко очерченных границ ответственности.

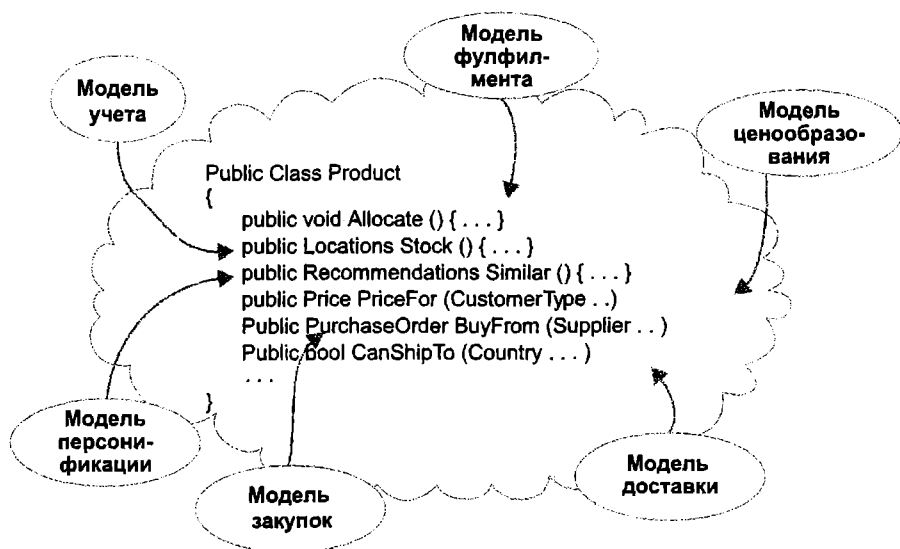


Рис. 6.6. Класс Product, реализация антишаблона «Божественный объект»

Интеграция с унаследованным или сторонним кодом

Еще одной причиной использования более мелких моделей является интеграция с унаследованным (существующим) или сторонним кодом. Добавление новых особенностей в монолитный код может оказаться трудоемким делом, когда в системе имеется большой объем унаследованного кода. Представьте, что добавляете новую, ясную, интуитивно понятную модель, созданную совместно со специалистами в предметной области, но ограничения унаследованного кода мешают ее выразительности. Однако когда имеется множество мелких моделей, не все из них должны будут взаимодействовать с унаследованным кодом.

Множество шаблонов, обсуждаемых в главе 11, показывают, насколько просто применяются принципы DDD к унаследованным системам, когда проект разбит на множество мелких моделей.

Предметная модель не является моделью предприятия

В некоторых случаях единая модель для всей системы может оказаться предпочтительнее, например, для реализации информационной системы и системы отчетов. Однако модель предприятия не лучший выбор для создания гибкой предметной модели, которая явно выражает предметные понятия. Также модель предприятия плохо подходит для итеративного процесса разработки, направленного на регулярное выявление новых ценных особенностей бизнеса.

На рис. 6.7 показано, как можно взять лучшее из двух миров — определить уникальную модель для каждого контекста и модель предприятия для информационной системы.

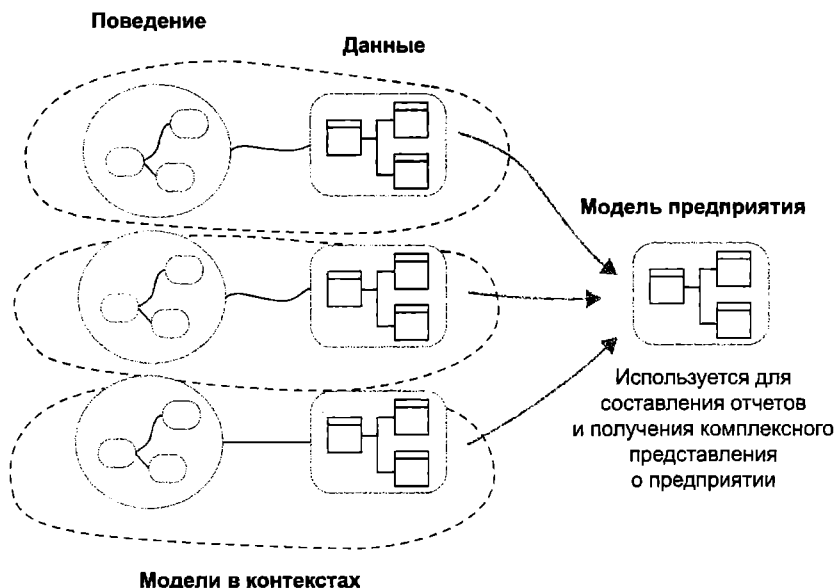


Рис. 6.7. Различия между предметной моделью и моделью предприятия

ПРИМЕЧАНИЕ

Во второй и третьей частях книги рассматриваются стратегии, такие как «публикация/подписка» (publish/subscribe), которые можно использовать для передачи данных из ограниченных контекстов в хранилище, чтобы на их основе создать модель предприятия.

Использование ограниченных контекстов для декомпозиции больших моделей

Ограниченный контекст определяет область применения модели. Он ясно дает понять, для каких целей используется модель, каким задачам она должна соответствовать и какими данными может пренебречь. Ограниченный контекст гарантирует, что предметные понятия за пределами контекста модели не будут отвлекать от задач, для решения которых она предназначена. Ограниченный контекст явно сообщает группам разработчиков, за что отвечает эта модель, а за что не отвечает.

Контекст — важное понятие в предметно-ориентированном проектировании. Каждая модель имеет контекст, неявно определяемый внутри области. Говоря о продукте в контексте области фулфилмента, нет необходимости пояснять, что речь идет о продукте, с которым можно производить операции фулфилмента; аналогично, в контексте продаж не обязательно говорить, что речь идет о продаваемом продукте. Это просто продукт в определенном контексте.

Общаясь со специалистами в предметной области или с другими разработчиками, убедитесь, что ваши собеседники понимают, в каком контексте ведется диалог.

Контекст определяет область действия модели, ограничивает границы предметного пространства, позволяет разработчикам сосредоточиться на главном.

В главе 4 «Проектирование на основе модели» вы познакомились с понятием единого языка (Ubiquitous Language, UL) и важностью определения моделей в контексте, лишенном лингвистической неоднозначности. Контекст определяет область ответственности модели, что помогает разделить и организовать предметное пространство. Ограниченный контекст развивает идею модели в контексте еще дальше, заключая ее в определенные границы ответственности. Эти границы являются конкретной технической реализацией в отличие от понятия контекста, более абстрактного по своей природе. Ограниченный контекст вынуждает выстраивать коммуникацию таким образом, чтобы не преуменьшать ясность модели.

Ограниченный контекст — это прежде всего лингвистическая граница. Если в процессе общения со специалистом вы чувствуете, что некоторая фраза требует уточнения контекста, это явный намек на то, что модель желательно изолировать в ограниченном контексте.

ОГРАНИЧЕННЫЙ КОНТЕКСТ = ПОГРАНИЧНЫЙ КОНТРОЛЬ

Рассматривайте ограниченные контексты как границы страны. Ничто не должно проникать внутрь ограниченного контекста без проверки на границе. Программный код в ограниченных контекстах можно сравнить с людьми, живущими в разных странах и говорящими на разных языках. Будьте настороже с людьми, которые пытаются проникнуть через ваши границы, не соблюдают ваши правила и не говорят на вашем языке. Одним из важнейших разделов DDD является защита границ. Модель определяется в контексте. Это правило должно неукоснительно соблюдаться в программной реализации, иначе вы получите большой «ком грязи». На рис. 6.8 продолжается пример демонстрации неоднозначности понятия «продукт»; здесь показано понятие продукта, существующее за рамками явно определенного контекста. Оно было сильно деформировано, чтобы удовлетворить потребностям самых разных сценариев использования. Без четкого обозначения границ модели и определения ее в конкретном контексте вы получите массу беспорядочного кода.

На рис. 6.9 показано, как понятие «продукт» можно сделать более концентрированным, применяя его в определенном контексте. При разработке приложения важно изолировать модели в ограниченных контекстах, чтобы избежать размывания обязанностей и не оказаться с большим «комом грязи».

ОРГАНИЗАЦИЯ КОДА ИМЕЕТ ЗНАЧЕНИЕ

Как разработчик, вы должны уделять большое внимание организации программного кода, чтобы иметь возможность управлять решениями для сложных предметных областей. Ограниченные контексты помогают организовать код на макроуровне, и это их свойство вы должны оценивать как особенно важное.

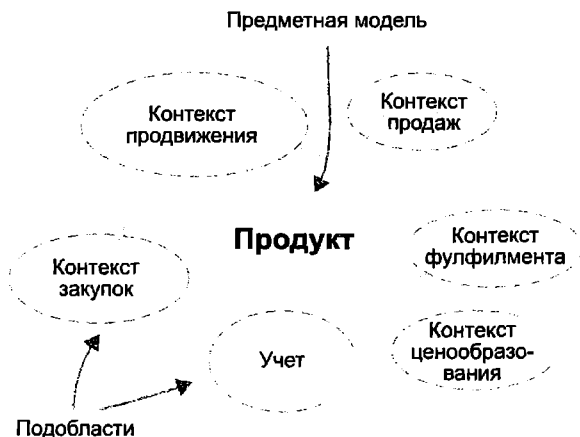


Рис. 6.8. Заключение терминов в контекст и идентификация множества моделей



Рис. 6.9. Определение моделей внутри собственных контекстов

Определение границ модели

Необходимость применения ограниченных контекстов в больших системах очевидна, но сам процесс идентификации ограниченных контекстов и их границ может вызывать сложности. К счастью, это решение не обязательно должно быть безошибочным сразу же. По мере знакомства с предметной областью вы сможете корректировать границы своих контекстов.

Существует два аспекта предметной области, которые можно использовать для идентификации ограниченных контекстов, — терминология и бизнес-потенциалы компании. Как объяснялось ранее в этой главе, один и тот же термин может иметь

разные значения в разных контекстах. Если вы можете изложить предметную модель по-другому, в зависимости от изменений значения слова или фразы, это с большой долей вероятности означает, что вы определили границу ограниченного контекста. Бизнес-потенциалы предприятия часто легко различимы, но иногда могут вводить в заблуждение. Например, если на предприятии имеется отдел продаж и отдел обслуживания покупателей, это почти наверняка говорит о необходимости создания соответствующих ограниченных контекстов. Но так бывает не всегда, поэтому важно не копировать слепо структуру предприятия.

Помимо предметной области, большое влияние на границы контекстов могут оказывать организация и местонахождение коллектива разработчиков, а также необходимость интеграции с унаследованными или сторонними системами.

А вот размер не играет решающей роли в определении ограниченных контекстов. Нет никаких явных или неявных признаков, по которым можно было бы судить, сколько классов или строк кода потребуется. Размер ограниченного контекста зависит в основном от аспектов предметной области. Некоторые ограниченные контексты могут быть очень большими, а другие — очень маленькими.

Таким образом, на границы контекстов могут оказывать влияние:

- неоднозначность терминов и предметных понятий;
- ориентация на подобласти и бизнес-потенциалы компании (функционал);
- организация и физическое местонахождение групп разработчиков;
- необходимость интеграции с унаследованным кодом;
- необходимость интеграции со сторонними системами.

ПРИМЕЧАНИЕ

Во второй части будут представлены практические примеры деления отдельных ограниченных контекстов на более мелкие модули и компоненты с сохранением представления предметной области.

Определение границ по языку

Беседуя со специалистами в предметной области, очень важно четко обозначить контекст беседы, потому что в разных контекстах одни и те же термины могут иметь разный смысл. Как неоднократно повторялось в этой главе, в процессе работы над вашей предметной областью будет использоваться несколько моделей. Для сохранности достоверности терминов необходимо провести четкие лингвистические границы. Соответственно лингвистические границы совпадают с границами ограниченных контекстов. Если, к примеру, понятие «продукт» имеет несколько значений в одной и той же модели, такую модель следует разделить как минимум на два ограниченных контекста, в каждом из которых будет действовать свое определение понятия «продукт». Этот вопрос уже обсуждался выше и иллюстрировался на рис. 6.9. Аналогично, один и тот же термин может ссылаться на несколько понятий, как в случае с заявкой, изображенной на рис. 6.3. Это еще один пример лингвистической границы, которая должна стать границей ограниченного контекста.

Учет бизнес-потенциалов компании

Любая организация — это экосистема взаимозависимых служб, каждая из которых имеет свой словарь терминов. Следовательно, бизнес-потенциалы компании являются яркими индикаторами лингвистических границ. Как отмечалось выше, отделы продаж и обслуживания покупателей могут иметь совершенно разные толкования понятия «заявка». Соответственно организационное деление компании следует рассматривать как вероятные границы контекстов.

ПРИМЕЧАНИЕ

Во второй части показано, как можно использовать сервис-ориентированные архитектуры (Service Oriented Architecture, SOA) для создания служб, соответствующих бизнес-потенциалам компании.

Будьте осторожны, используя организационную структуру компании для определения ограниченных контекстов. Иногда организационная структура не в полной мере согласуется с предметной областью. В конечном итоге вы рискуете получить систему, отражающую структуру взаимодействий подразделений вместо точного представления предметной области. Закон Конвея даже явно указывает, что система неизменно отразит структуру взаимодействий подразделений:

«Организация, разрабатывающая систему (слово «система» подразумевается здесь в более широком понятии, чем «информационная система»), неизбежно воспроизведет структуру, повторяющую состав взаимодействий внутри организации».

Закон Конвея можно использовать как руководство в двух направлениях. Прежде всего, можно, зная закон Конвея, постараться не воспроизвести структуру организации. А можно реконструировать предприятие, опираясь на желаемую архитектуру системы. Любой из этих подходов требует значительных усилий, поэтому такие решения нельзя принимать без тщательного планирования.

ПРИМЕЧАНИЕ

Что фактически представляют собой бизнес-потенциалы компании? Бизнес-потенциалы — это группы сотрудников предприятия, осуществляющих бизнес-процессы, в свою очередь разделенные на более мелкие бизнес-процессы. Представьте бизнес-потенциал фулфилмента: его составляет менеджер (менеджеры), управляющий работой сотрудников склада, вкупе с самими сотрудниками склада, осуществляющими более низкоуровневые процессы, такие как упаковка и рассылка товаров, и тем самым вносящими свой вклад в осуществление бизнес-процесса более высокого уровня — выполнения заказа.

Создание контекстов на основе организации разработчиков

За ограниченный контекст должна отвечать одна группа разработчиков, независимо от того, охватывает он несколько приложений или подразделений или только одно. Таким образом, группы разработчиков должны быть организованы согласно границам ограниченных контекстов; сформируйте группы управления продуктом

и обслуживанием, а не пытаетесь повторить организационную структуру предприятия. Убедитесь, что границы ответственности групп совпадают с границами ограниченных контекстов, от уровня представления до предметной логики и хранения данных.

ПРАВИЛО ДВУХ ПИЦЦ В AMAZON

В компании Amazon существует правило, согласно которому группа разработчиков не должна быть настолько большой, чтобы ей не хватило двух пицц на обед (<https://www.insight-it.ru/highload/2008/arkhitektura-amazon/>). Очень важно, чтобы группы разработчиков оставались небольшими, сосредоточенными на решении узкого круга задач и отвечали за один или несколько ограниченных контекстов. Как демонстрирует карта контекстов, не все контексты действуют в изоляции. Кроме того, существуют специальные шаблоны организации взаимодействий между ограниченными контекстами и шаблоны взаимодействий групп разработчиков.

Главным обоснованием необходимости организации групп разработчиков в соответствии с границами контекстов является независимость, позволяющая группам двигаться быстрее и принимать более удачные решения. Группы могут двигаться вперед быстрее, когда полностью контролируют продукт и технические решения. Также они могут быстрее совершать итерации, если не приходится беспокоиться о влиянии на другие группы.

За каждое направление, важное для предприятия, может отвечать одна отдельная группа разработчиков, чтобы при необходимости принять решение или с появлением у кого-либо интересного предложения все могли быстро собраться и решить, куда двигаться дальше. С другой стороны, разные группы могут отвечать за разные направления и зависеть от возможности эффективного сотрудничества. Например, одна группа может быть вынуждена ждать, пока другая освободится для встречи, на которой можно будет принять решение.

Помните, взаимодействия между группами — это очень даже хорошо, поэтому не избегайте их полностью, лишь ограничьте случаями, когда эти взаимодействия действительно пойдут на пользу. Примером таких полезных взаимодействий групп разработчиков может служить обмен знаниями и навыками.

Старайтесь сохранить взаимодействие между группами

Даже при том, что полная независимость групп обеспечивает наивысшую продуктивность, важно сохранить общение между группами для обмена знаниями и навыками. В конечном счете, ограниченные контексты объединяются во время выполнения и все вместе обеспечивают полное соответствие сценариям использования, поэтому группы должны иметь общее представление о месте своих ограниченных контекстов в общей картине системы. С этой целью во многих организациях проводят регулярные встречи, во время которых разработчики делятся информацией о том, над чем они работают, как реализуют решение тех или иных задач, какие технологии помогают им в достижении целей. Иногда используется

такой замечательный прием, как обмен разработчиками, когда разработчик переходит на несколько дней в другую группу, чтобы поближе познакомиться с разделом предметной области, над которой работает эта группа. Вы можете придумать свои, новые подходы, основанные на описанных выше — встречах групп и обмене разработчиками.

Усилия, направленные на повышение эффективности взаимодействий между группами разработчиков, с лихвой окупятся, когда возникнет необходимость важных перемен. Фактически вы получите выгоды в любой системе. В некоторый момент контракт между ограниченными контекстами понадобится изменить, чтобы привести в соответствие с потребностями предприятия. В таком случае организация взаимодействия групп для выработки наиболее удачного решения может оказаться самым эффективным вариантом.

ПРИМЕЧАНИЕ

Сохранение обратной совместимости также является эффективным подходом, помогающим избежать необходимости крупных изменений (и налаживания тесного взаимодействия групп). Подробнее о сохранении обратной совместимости рассказывается во второй части книги, включая обмен сообщениями и примеры на основе архитектуры REST.

Диаграммы и краткая документация помогают группам быстро обмениваться знаниями, что особенно ценно, когда в группу вливаются новые члены. В главе 7 «Карты контекстов» вы увидите, как карты контекстов и другие диаграммы упрощают передачу знаний.

Игра «Контексты»

Чтобы лучше осознать важность моделирования с применением контекстов и определения множества моделей в предметной области, вы можете опробовать вспомогательную игру. Игра «Контекст» (Context Game), придуманная Греггом Янгом (<http://codebetter.com/gregyoung/2012/02/29/the-context-game-2/>), поможет прояснить, где необходимо ввести дополнительные модели в отображение предметного пространства.

Игру можно включить в процедуру переработки знаний, когда возникнет подозрение, что появился какой-то перегруженный или неоднозначный термин. Разделите команду на небольшие группы специалистов и разработчиков. Специалистов лучше разделить по принадлежности к отделам на предприятии или по областям их ответственности. Дайте им 20 минут, чтобы сформулировать определение термина или понятия, как оно используется в их подразделении, затем подключите к обсуждению разработчиков для усвоения новой информации. Затем соберите всю команду вместе и обсудите их взгляды на термин или понятие.

Вы увидите, что в разных подразделениях предприятия одни и те же термины могут иметь разные значения. Там, где на предприятии обнаруживаются различия в определении терминов, вы должны провести границу контекста и создать новую модель. Это было показано на рис. 6.8 в отношении понятия «продукт», существующего в нескольких разных контекстах.

Различия между подобластями и ограниченными контекстами

Области (или подобласти), о которых рассказывалось в главе 3 «Концентрация на смысловом ядре», представляют логические разделы всей предметной области и часто отражают организационную структуру предприятия. Они используются для отделения важных разделов приложения, смыслового ядра, от менее важных, неспециализированных областей (generic domains) и областей поддержки (supporting domains). Назначение областей — дистилляция предметного пространства и уменьшение сложности.

Предметные модели создаются с целью удовлетворения сценариям использования в каждой из областей. В идеале каждой области должна соответствовать своя модель, но так бывает не всегда. Модели определяются исходя из организации групп разработчиков, неоднозначностей в языке, особенностей бизнес-процессов или физической разбросанности подразделений предприятия. Поэтому область может содержать несколько моделей, а одна модель может охватывать несколько областей. Это часто наблюдается в унаследованных окружениях.

Модели должны быть изолированными и определяться внутри явно обозначенного контекста, чтобы оставаться ясными и неразмытыми. Как вы уже знаете, такой контекст называют ограниченным контекстом (bounded context). В отличие от подобласти, ограниченный контекст является конкретной технической реализацией, устанавливающей границы между моделями в приложении. Ограниченные контексты существуют в пространстве решений и явно представляют собой предметные модели в контексте.

Реализация ограниченных контекстов

Ограниченному контексту принадлежит вертикальный срез функциональности, от уровня представления до уровня предметной логики и даже логики хранения данных.

Применение концепции ограниченных контекстов к системе, изображенной ранее на рис. 6.5, дает в результате систему, в которой каждый ограниченный контекст отображает свою информацию посредством собственного уровня представления, выполняет свою предметную логику и сохраняет свои данные в хранилище, как показано на рис. 6.10. В такой улучшенной архитектуре понятие «продукт» может существовать в каждом ограниченном контексте и обладать только атрибутами и логикой, соответствующими данному контексту. Изменения в любом ограниченном контексте больше не оказывают нежелательного влияния на другие ограниченные контексты, потому что области теперь изолированы друг от друга.

Детальное исследование (рис. 6.11) показывает, что понятие «продукт» существует в двух моделях, но определяется контекстом, в котором оно находится.

Не все ограниченные контексты должны следовать одному и тому же архитектурному шаблону. Если ограниченный контекст содержит неспециализированную область или область поддержки с несложной логикой, может оказаться предпочтительным реализовать такой ограниченный контекст в стиле CRUD. Но если

Приложение электронной коммерции

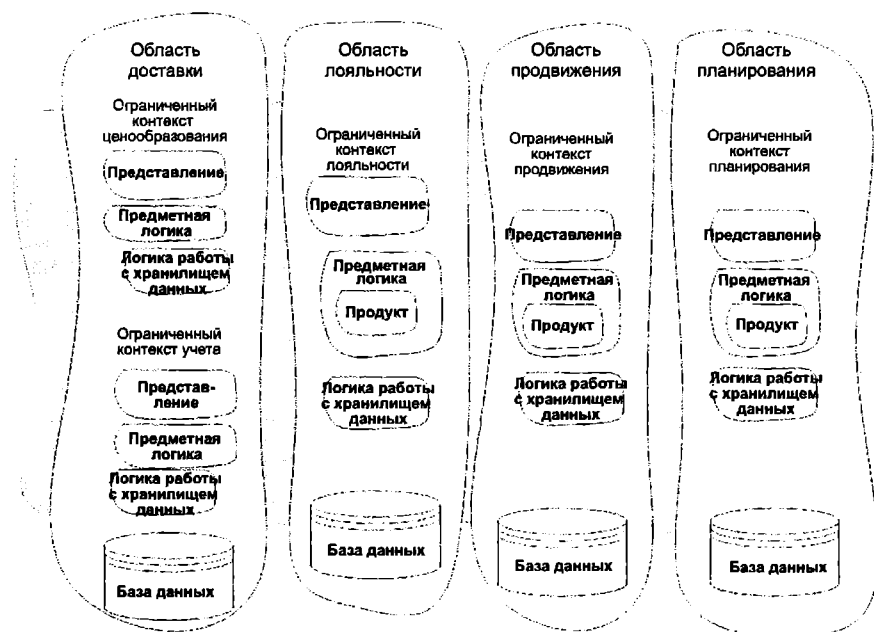


Рис. 6.10. Шаблон многоуровневой архитектуры для каждого ограниченного контекста

предметная логика чрезвычайно сложна, для такой области лучше определить полнофункциональную объектно-ориентированную модель. После отделения ограниченных контекстов можно сделать еще шаг вперед и применить архитектурные шаблоны, как показано на рис. 6.12.

ЧТО ТАКОЕ CRUD?

CRUD — аббревиатура, образованная из слов Create, Read, Update и Delete (создание, чтение, изменение и удаление). Этой аббревиатурой часто описывают системы с небольшим объемом логики, реализующей модель данных. Для многих разработчиков CRUD — всего лишь слово из четырех букв (впрочем, это действительно так), и они думают, что слишком простое решение ниже их достоинства. Не бойтесь использовать в приложениях архитектуры в стиле CRUD. Если вы имеете дело с ограниченным контекстом, не содержащим логики, нет смысла нагромождать слои абстракций. Помните о принципе «to KISS» («Keep it simple, stupid» — «Делай проще, дурачок»).

На рис. 6.12 показано, как можно использовать разные архитектурные шаблоны в разных ограниченных контекстах внутри приложения. Разные ограниченные контексты взаимодействуют посредством составного пользовательского интерфейса для предоставления информации пользователю. Рисунок 6.13 показывает, что ограниченный контекст инкапсулирует инфраструктуру, логику работы с хранилищем данных и интерфейс с пользователем, а также предметную модель.

Приложение электронной коммерции

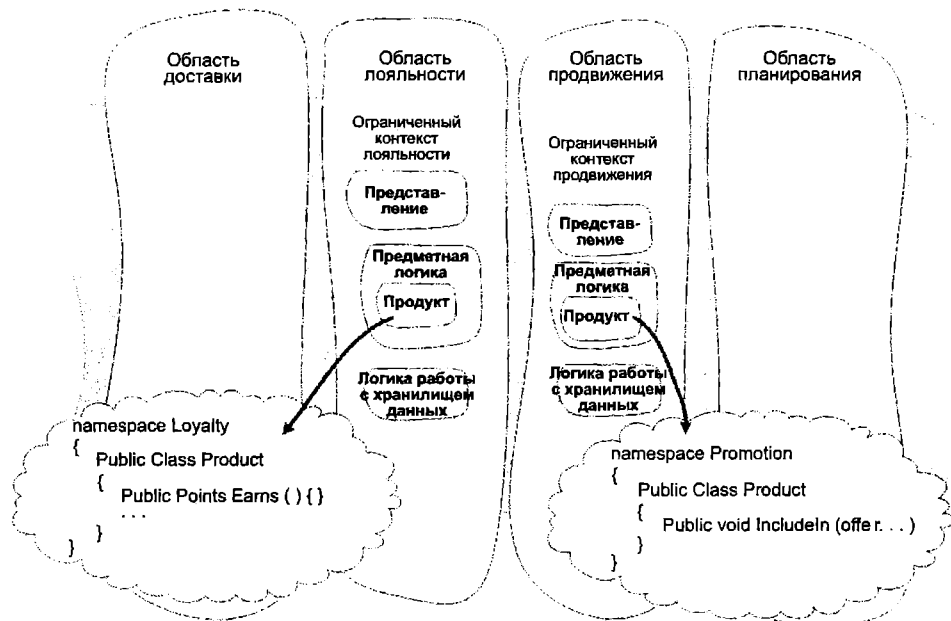


Рис. 6.11. Класс Product в разных контекстах

ЧТО ТАКОЕ CQRS?

CQRS — аббревиатура, образованная из слов Command Query Responsibility Segregation (разделение ответственности команд и запросов). Это архитектурный шаблон, отделяющий обработку запросов от обработки команд путем определения двух моделей вместо одной. Одна модель создается для обработки команд, а другая — для нужд уровня представления. Подробно этот шаблон будет рассматриваться в главе 24 «CQRS: архитектура ограниченного контекста».

ЧТО ТАКОЕ СОСТАВНОЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

Составной интерфейс пользователя — это пользовательский интерфейс, состоящий из слабосвязанных компонентов. Составной интерфейс отображает на одной веб-странице данные из нескольких ограниченных контекстов. Это может достигаться, например, путем выполнения нескольких Ajax-запросов. Для защиты целостности модели ограниченный контекст может посредством прикладных служб экспортировать методы, скрывающие детали соответствующей предметной модели, как показано на рис. 6.12. Такая конечная точка может принимать исходные данные из разных моделей и переводить их на язык, понятный модели. Это обеспечит защиту целостности модели и поможет предотвратить размытие границ ответственности моделей.

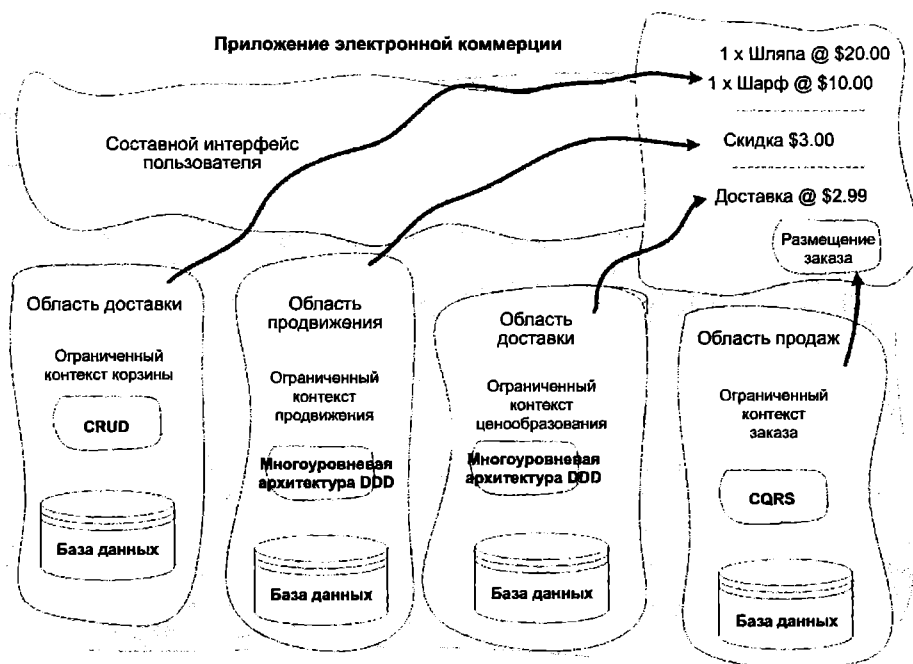


Рис. 6.12. К разным ограниченным контекстам можно применять разные архитектурные шаблоны

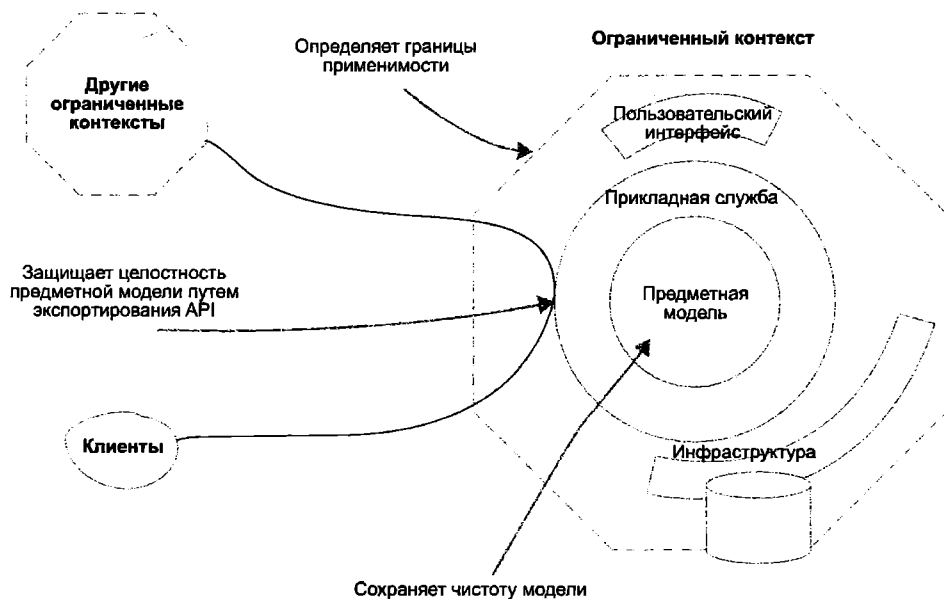


Рис. 6.13. Строение ограниченного контекста

ПРИМЕЧАНИЕ

Подробнее о прикладных службах рассказывается в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования».

По сути, автономность является ключевой характеристикой ограниченных контекстов, изолирующей группы разработчиков от внешних влияний и изолирующей модели от проникновения в них нежелательных понятий. Во второй части вы увидите практические примеры реализации и интеграции автономных ограниченных контекстов с использованием масштабируемых приемов интеграции, включая архитектуру, управляемую сообщениями (event-driven architecture), поддерживающую обмен сообщениями и REST.

Ключевые идеи

- Попытка использовать единую модель для представления сложной предметной области часто приводит к появлению программного кода, напоминающего большой «ком грязи».
- Монолитные модели требуют дополнительных затрат на организацию взаимодействий между группами разработчиков и снижают их эффективность в разработке и развертывании новых функциональных особенностей, ценных для предприятия.
- Для каждой модели в приложении необходимо явно определить ее контекст и сообщить его другим группам разработчиков.
- Ограниченный контекст — это лингвистическая граница. Он изолирует модели, чтобы избежать неоднозначности в едином языке.
- Ограниченный контекст защищает целостность предметной модели.
- Идентификация и создание ограниченных контекстов является одним из важнейших аспектов предметно-ориентированного проектирования.
- Нет никаких правил определения границ моделей и, соответственно, ограниченных контекстов. Ограниченные контексты должны создаваться на основе лингвистических границ, организации и физического местонахождения групп разработчиков.
- Подобласти используются в предметном пространстве для разделения всей предметной области. Ограниченные контексты используются для определения применимости модели в пространстве решений.
- Ограниченным контекстом должна заниматься только одна группа разработчиков.
- Архитектурные шаблоны применяются на уровне ограниченного контекста, но не на уровне приложения. Если ограниченный контекст не содержит сложной логики, для его реализации можно с успехом использовать архитектуру CRUD.
- В рамках ограниченного контекста следует разговаривать на едином языке.
- Ограниченный контекст должен быть автономным — он должен включать весь программный стек, от представления до предметной логики и логики работы с базой данных.

7

Карты контекстов

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Почему карта контекстов имеет большое значение для стратегического проектирования
- Модель отношений между ограниченными контекстами
- Шаблоны организационных отношений между группами разработчиков и контекстами
- Эффективные способы передачи карт контекстов

В больших и сложных приложениях множество моделей в контексте взаимодействуют друг с другом, обеспечивая требуемое поведение системы. Компоненты системы могут разрабатываться не одной группой разработчиков. Некоторые компоненты могут быть унаследованы от прежних систем, за разработку которых отвечали другие группы, другие компоненты могли быть предоставлены третьими сторонами, не имеющими никакого представления о клиентах, которые будут пользоваться их программным кодом. Группы, не имеющие хорошего представления о различных контекстах в системе и отношениях между ними, рискуют нарушить целостность моделей при интеграции ограниченных контекстов. Границы между моделями могут оказаться стертыми, и в результате получится большой «ком грязи», если группы разработчиков не зафиксируют отношения между контекстами и не разберутся с ними.

Технические детали контекстов не единственная сила, которая может помешать успеху проекта. Организационные отношения между группами разработчиков, ответственными за разные контексты, также могут существенно сказаться на результате проекта. Часто группы, занимающиеся разработкой других контекстов, имеют другие мотивации или другие приоритеты. Для успеха проекта группы обычно должны управлять изменениями не на техническом, а на организационном уровне.

В ходе разработки могут возникать другие проблемы нетехнического характера. К ним относятся проблемы, проистекающие из разделов предметной области, которые находятся между ограниченными контекстами и не были явно определены.

Некоторые бизнес-процессы часто находятся вне зоны ответственности групп разработчиков и самого предприятия, но, как это ни парадоксально, оказываются чрезвычайно важны для предприятия и его бизнес-процессов.

Для преодоления этих проблем группы разработчиков могут создавать карты контекстов, отображающие технические и организационные отношения между разными ограниченными контекстами. Самое главное достоинство карты контекстов заключается в отображении ландшафта реальности, со всеми подробностями, в противоположность устаревшей высокоуровневой проектной документации. Карта контекстов, постоянно развивающаяся и уточняющаяся, гарантирует, что группы разработчиков будут иметь целостное представление о системе, как техническое, так и организационное, что даст им прекрасный шанс преодолеть проблемы на ранних этапах и избежать снижения полезности моделей из-за нарушения их целостности.

Карта реальности

Карта контекстов, как показано на рис. 7.1, является важным артефактом; она отвечает за явное определение границ разных контекстов в системе и помогает группам разработчиков видеть точки контакта между ними. Карта контекстов не является подробным документом, созданным с помощью некоторого инструмента проектирования уровня предприятия, это обобщенная диаграмма, нарисованная вручную, которая отражает общую картину контекстов, задействованных в разработке. Карта контекстов должна быть достаточно простой, чтобы ее понимали и специалисты в предметной области, и сами разработчики. Помимо маркировки контекстов, понятной разработчикам, диаграмма должна также отображать малопонятные области системы, чтобы отразить некоторые неприглядные реалии программного кода.

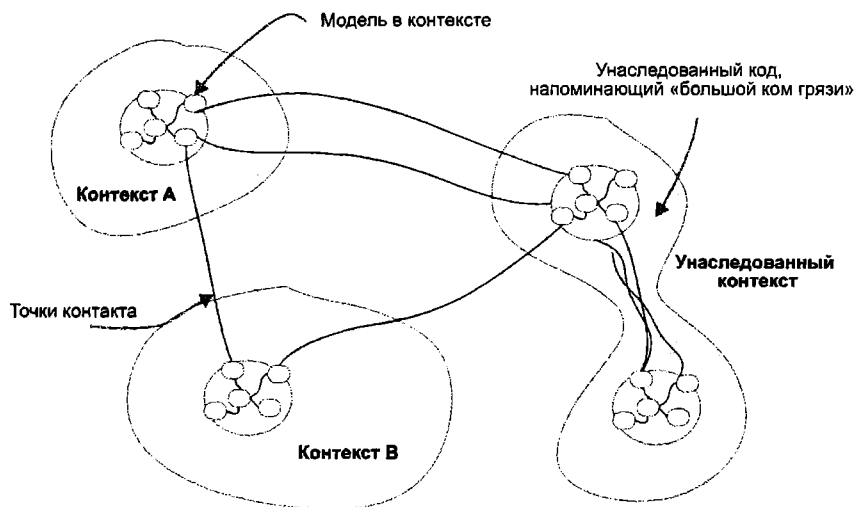


Рис. 7.1. Карта контекстов

Техническая действительность

Технические детали карты, как показано на рис. 7.2, демонстрируют точки интеграции между контекстами. Эта карта жизненно важна для понимания разработчиками технических следствий их изменений. Она показывает существующие границы и любые преобразования, используемые для сохранения целостности ограниченных контекстов.

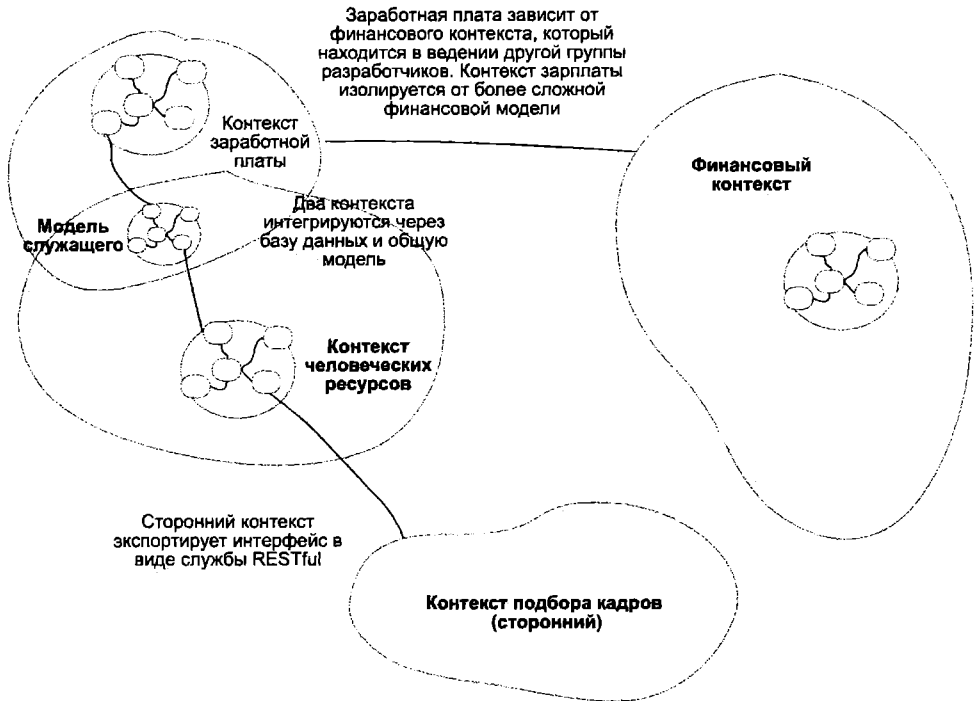


Рис. 7.2. Технические детали интеграции на карте контекстов

Очень важно, чтобы карты контекстов отражали действительность, показывали текущее состояние реализации, а не идеализированное будущее. Карты контекстов не должны содержать детального описания моделей, но должны показывать точки интеграции и потоки данных между ограниченными контекстами. Подобно программной и аналитической моделям, карта контекстов должна изменяться, но только с изменением программного кода, чтобы не создавать ложного представления о ландшафте. Карта должна абсолютно соответствовать действительности, только тогда она будет полезна.

Организационная действительность

Изменения в имеющихся бизнес-процессах или создание новых часто могут охватывать множество ограниченных контекстов и затрагивать разные части пред-

метной области. Координация подобных масштабных изменений часто требует не менее масштабного управления группами разработчиков для синхронного внесения технических изменений в реализацию. Важно знать, кто отвечает за каждый контекст, для которого требуется внести изменения, и как эти изменения будут внесены. Если процесс координации и очередности изменений не проработан до конца, это может стать камнем преткновения и вызвать остановки в разработке, так как одни группы будут вынуждены ждать, пока другие внесут свои изменения. Отображение векторов взаимодействия между группами разработчиков — одно из преимуществ карты контекстов перед традиционными архитектурными диаграммами и диаграммами UML. Понимание этого процесса до начала работы над проектом — это ключ к разрешению нетехнических проблем до того, как они блокируют процесс разработки.

На рис. 7.3 изображены направления связей между ограниченными контекстами. Группы разработчиков, работающие над разными проектами, наверняка осознают, что план релизов должен согласовываться с порядком разработки при возникновении необходимости внести изменения в ограниченный контекст, находящийся вне зоны их ответственности. Техническая реализация изменений может оказаться очень простой, но если ситуация взаимоотношений непонятна, изменения в других контекстах могут отставать или вообще оставаться нереализованными.

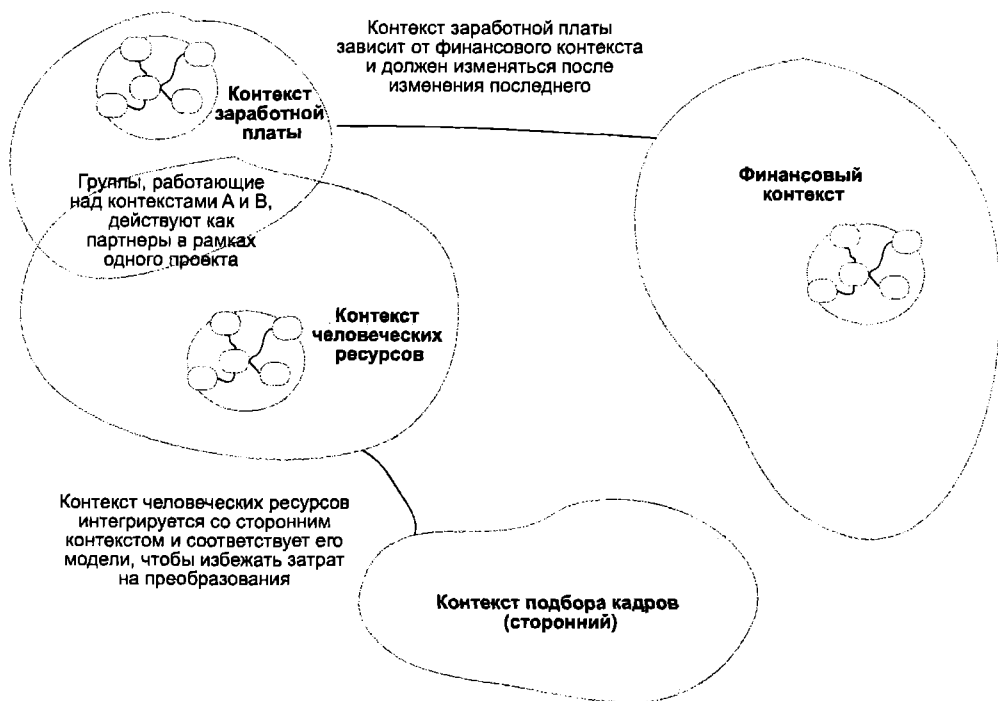


Рис. 7.3. Отражение организационных взаимодействий на карте контекстов

Отображение актуальной действительности

Создавая карту контекстов, постарайтесь сосредоточиться непосредственно на своей предметной области; вам необходимо понять, какой ландшафт приведет к успеху вашего конкретного проекта, а не предприятия в целом. Сосредоточенность на контекстах, с которыми будет происходить прямая интеграция, поможет наладить процесс составления карты и удержать в фокусе главную задачу.

Выделение смыслового ядра на карте

В процессе картирования контекстов и идентификации моделей обязательно работайте со специалистами в предметной области и наметьте смысловое ядро. Выделение смыслового ядра на карте и определение связей смыслового ядра с другими контекстами может помочь прояснить его значение в контексте ландшафта предприятия.

Определение отношений между ограниченными контекстами

Модели в контекстах действуют в больших приложениях сообща, обеспечивая требуемое поведение системы. Важно понять, какими отношениями связаны контексты, чтобы получить более полное представление о текущей ситуации. Шаблоны, о которых рассказывается ниже, описывают отношения между ограниченными контекстами. Обратите внимание, что эти шаблоны показывают, как связаны модели и как — группы разработчиков. Они не являются техническими шаблонами интеграции контекстов — техническая сторона интеграции ограниченных контекстов будет рассматриваться во второй части книги.

Предохранительный слой

Если создается модель для подсистемы, взаимодействующей с другими подсистемами в составе большой системы, может понадобиться реализовать интерфейс для моделей, созданных разными группами разработчиков. Другие модели, даже созданные для той же предметной области, могут выражаться на другом едином языке и моделироваться совершенно иначе, чем принято у вас. Если не проявить осторожность при интеграции с такими моделями, адаптация под их интерфейсы может привести к повреждению вашей модели.

Чтобы избежать повреждений и защитить модель от внешнего влияния, можно создать изолирующий слой (уровень), содержащий интерфейс, реализованный по правилам вашей модели. Интерфейс будет выполнять адаптацию и трансляцию под интерфейс другого контекста. Такой изолирующий слой часто называют предохранительным слоем (anticorruption layer).

Как показано на рис. 7.4, предохранительным слоем можно обернуть взаимодействия с унаследованным или сторонним кодом для защиты целостности ограниченного контекста. Предохранительный слой управляет преобразованием данных из представления в одном контексте в представление в другом.

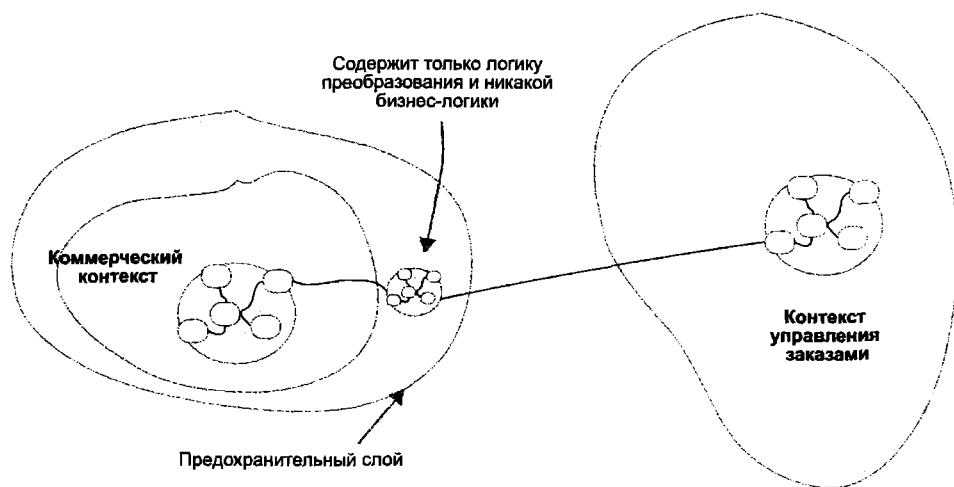


Рис. 7.4. Используйте предохранительный слой для интеграции со сторонним или неизменяемым кодом

Предохранительный слой действует подобно шаблону «Адаптер» — он преобразует прикладной интерфейс (API) одного контекста в прикладной интерфейс другого. В главе 11 приводится пример таких преобразований между ограниченными контекстами с использованием предохранительного слоя.

В реальной жизни не всегда приходится заниматься разработкой совершенно новых систем, иногда бывает необходимо интегрировать свой код со сторонними или унаследованными контекстами. Так как API этих контекстов обычно недоступны для изменения или их изменение сопряжено с большими сложностями, важно не поставить под угрозу целостность вашего ограниченного контекста в попытках привести его в соответствие с API другого контекста.

Если имеется система, напоминающая большой «ком грязи», и необходимо добавить в нее дополнительные функциональные возможности, может показаться заманчивым просто добавить в нее новый код и... внести свою лепту в общую неразбериху. Как вариант, можно предложить переписать всю систему и предусмотреть в ней новые возможности. Однако ни один из этих вариантов нельзя назвать целесообразным, так как переделка большого приложения является рискованным занятием и может потребовать значительного времени, а простое добавление нового кода в имеющуюся «кашу» может еще больше усложнить сопровождение. Более практичным выглядит вариант, основанный на использовании предохранительного слоя (ПС) для изоляции нового контекста от унаследованного кода. Применение предохранительного слоя в этом случае даст вам великолепную практику рефакторинга, потому что вы сможете определить четкие границы без необходимости изменять запутанный код внутри этого контекста.

Общее ядро

Если две группы разработчиков при работе над одним и тем же приложением создают два разных контекста, имеющих множество пересечений в терминах

и логике предметной области, накладные расходы на изоляцию контекстов и использование преобразований из одного контекста в другой могут оказаться слишком большими. В такой ситуации более выигрышным может быть решение объединить усилия и сделать часть модели общей в целях упрощения интеграции. Такую общую модель обычно называют общим ядром (shared kernel). Данный шаблон особенно хорош, когда имеется два ограниченных контекста в одном разделе предметной области с общим подмножеством предметной логики.

На рис. 7.5 показана часть ERP-системы¹, содержащая контекст заработной платы и контекст человеческих ресурсов, которые используют общую модель служащего.

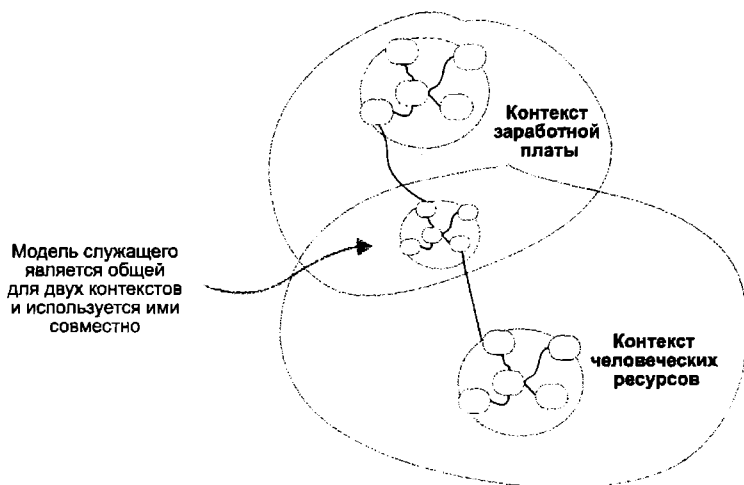


Рис. 7.5. Интеграция посредством общего ядра

Из-за возникновения зависимости от общего кода применение шаблона «Общее ядро» может оказаться рискованным, так как образуется тесная связь, которая может привести к тому, что изменения, внесенные одной группой, могут нарушить работу контекста, разрабатываемого другой группой. Поэтому важно, чтобы все члены обеих групп понимали это и использовали практику непрерывной интеграции для верификации поведения обеих моделей после внесения изменений в общую модель.

Служба с открытым протоколом

Другие системы или компоненты, взаимодействующие с вашим контекстом, также будут использовать некоторый слой, осуществляющий преобразование вашей модели в собственное представление подобно предохранительному слою. Если одна и та же логика преобразования используется множеством потребителей, может оказаться полезным предусмотреть набор служб, экспортирующих функциональность контекста посредством четко определенного контракта, известного как служба с открытым протоколом (open host service).

¹ ERP-система — интегрированная система управления предприятием. — *Примеч. пер.*

Взгляните на рис. 7.6. Система управления заказами предоставляет информацию о заказах клиента коммерческой системе, системе закупок и CRM-системе¹. Каждая система требует преобразования сложной модели заказа в собственное представление. Чтобы избежать дублирования кода, система управления заказами может экспортировать упрощенные версии заказов посредством службы с открытым протоколом, как показано на рис. 7.7.

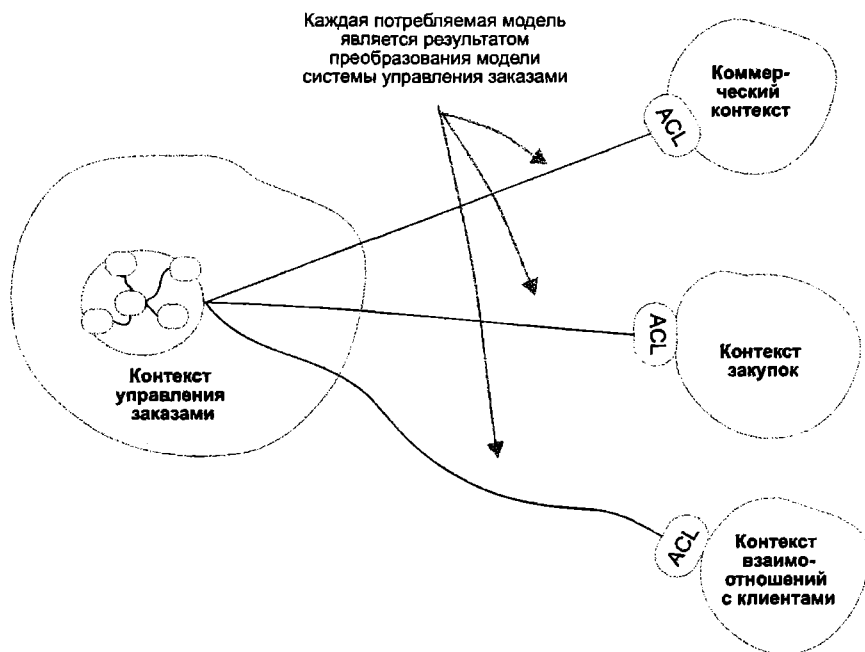


Рис. 7.6. Интеграция нескольких подсистем с одинаковыми требованиями

Отдельное существование

Если цена интеграции слишком велика из-за технических или организационных сложностей, можно вообще отказаться от интеграции контекстов и позволить группам разработчиков заниматься реализацией функций независимо друг от друга. Интеграция в этом случае может быть выполнена на уровне пользовательского интерфейса или вообще перенесена в разряд задач, выполняемых вручную. Например, в приложении обслуживания клиентов, которое управляет контактами с клиентами, может оказаться полезным показывать пользователям невыполненные заказы клиента. Однако если для интеграции с системой управления заказами потребуется приложить слишком много усилий, практичнее может оказаться простое включение ссылки в меню, с помощью которой пользователь сможет открыть систему управления заказами в отдельном окне и получить всю необходимую ему информацию без лишних сложностей, хотя и за счет некоторого снижения удобства.

¹ CRM-система — система управления взаимоотношениями с клиентами. — *Примеч. пер.*

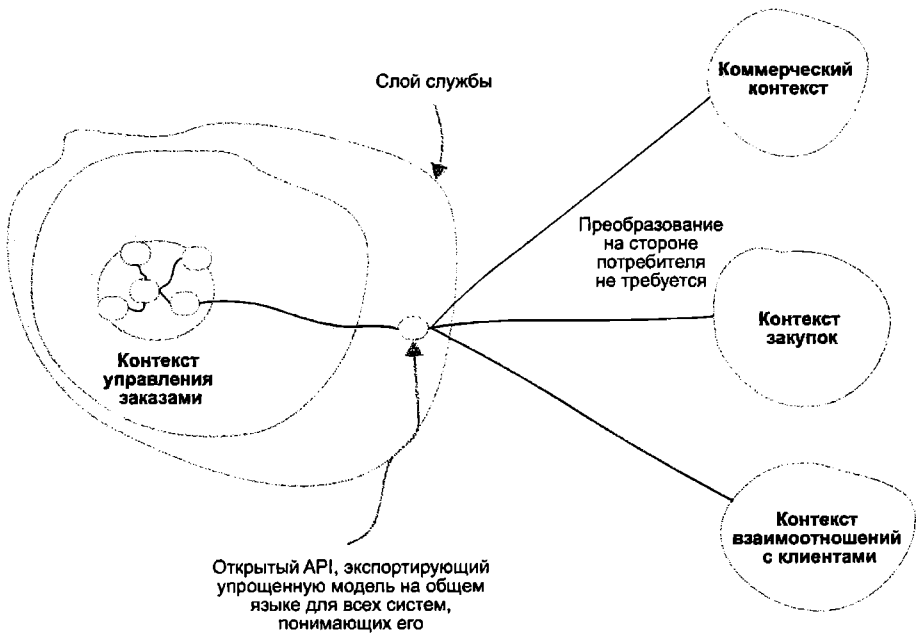


Рис. 7.7. Интеграция посредством службы с открытым протоколом

Партнерство

Если две группы разработчиков отвечают за разные контексты, но движутся к общей цели, можно организовать партнерство (partnership) двух групп, чтобы обеспечить интеграцию контекстов совместными усилиями. Совместными усилиями в этом случае можно разработать такие технические интерфейсы, которые будут отвечать интересам обеих групп. С организационной точки зрения может потребоваться согласовывать выпуски новых версий между группами, чтобы реализации всех интерфейсов и точек контактов выходили в нужное время. Если группы используют общее ядро между двумя ограниченными контекстами, им также можно порекомендовать организовать партнерские отношения между собой.

Отношения «вышестоящий/нижестоящий»

Иногда взаимодействие между ограниченными контекстами можно описать в терминах иерархии; одна сторона будет «вверху» (upstream), другая — «внизу» (downstream)¹. Если контекст находится «внизу», тогда будет наблюдаться зависимость ограниченного контекста от данных или поведения контекста, находящегося «вверху». Вышестоящий контекст будет влиять на нижестоящий. Например, если интерфейс верхнего контекста изменится, соответственно должен измениться

¹ Имеется в виду «вверху» и «внизу» по течению реки или потока: upstream и downstream соответственно. — *Примеч. пер.*

и нижний контекст. Аналогично, план выпуска новых версий верхнего контекста будет влиять на план выпуска новых версий нижнего контекста, так как последний может зависеть от определенного метода API. Описываемые далее шаблоны показывают, как можно классифицировать отношения «верхний/нижний».

Заказчик-поставщик

Если перед группами разработчиков стоят разные цели, то во избежание ситуации, когда все решения принимает «верхняя» группа и есть риск нарушить работу «нижней» группы, а также нанести ущерб проекту в целом, можно сформировать отношения вида «заказчик — поставщик» (customer-supplier). При использовании этого шаблона группы согласовывают интерфейс, удовлетворяющий техническим и организационным требованиям. Группа на стороне «заказчика» — это нижний контекст. Заказчик связывается с «поставщиком» (верхний контекст) и планирует встречи, чтобы удостовериться, что его потребности понятны «поставщику» и что он будет извещаться обо всех изменениях наверху.

Из-за необходимости дополнительных согласований в отношениях заказчик — поставщик на принятие решений может потребоваться больше времени. Группы должны организовывать встречи или видеоконференции, чтобы двигаться дальше. При внимательном отношении к организационным вопросам группы могут заранее договориться, что они не будут блокировать работу друг друга, заставляя одну группу ожидать, пока другая примет решение или развернет новую версию своей системы с обновленным интерфейсом. С другой стороны, если команды географически удалены друг от друга, находятся в разных часовых поясах или имеют плотные графики работы, организационные проблемы могут вызывать длительные простои.

Отношение «заказчик — поставщик» (customer-supplier) подчеркивает, что группа, разрабатывающая ограниченный контекст «заказчик», зависит от группы, разрабатывающей ограниченный контекст «поставщик», но не наоборот. Иногда нет никакой возможности организовать партнерские отношения с верхним контекстом, и по этой причине нижний контекст вынужден соответствовать точкам интеграции, определяемым верхним контекстом.

Например, взгляните на рис. 7.8: коммерческий контекст требует больше информации о заказе, чем передается в настоящий момент из контекста управления заказами. Группа, отвечающая за коммерческий контекст, может действовать как «заказчик» на встречах с командой, отвечающей за контекст управления заказами, постараться донести важность своих требований и обеспечить их удовлетворение.

Конформист

Если сотрудничество с верхним контекстом невозможно, тогда нижний контекст вынужден соответствовать точкам интеграции, определяемым верхним контекстом. В общем случае отношение «конформист» (conformist) напоминает отношения с внешними поставщиками. Практически ни один поставщик услуг платежных систем не пойдет на изменение своего API ради вас и не даст вам дополнительной информации, если только вы не являетесь особо влиятельным клиентом и не возобладаете над ними. Вместо этого, если вы находитесь внизу, не имеете

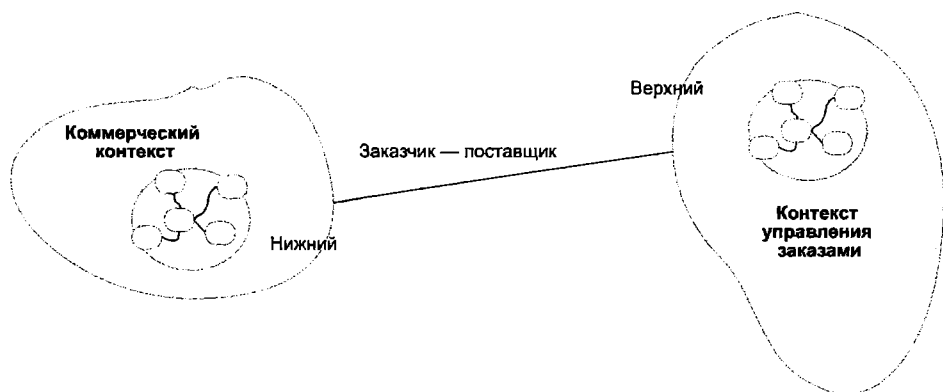


Рис. 7.8. Отношение «заказчик — поставщик» между ограниченными контекстами

возможности оформить отношения вида «заказчик — поставщик» и для вас слишком затратно создавать предохранительный слой, вам не остается ничего иного, как приспособиться к модели поставщика для упрощения интеграции. Наиболее очевидный недостаток конформистских отношений состоит в том, что команда, находящаяся внизу, вероятно, вынуждена будет пожертвовать ясностью своей предметной модели из-за необходимости приспособливаться под модель верхнего контекста, даже если это будет идти вразрез с их концептуальными представлениями. Как вариант, для сохранения целостности своей модели можно использовать предохранительный слой и устранить непосредственное влияние на вашу модель возможных изменений в контактной точке.

Отношения на карте контекстов

При составлении карты контекстов можно добавить в нее существующие организационные связи и указать типы интеграции между ограниченными контекстами над линиями, связывающими их. Можно также указать, какая сторона является верхней, а какая — нижней, используя буквы или символы. На рис. 7.9 показан пример карты контекстов с такими обозначениями.

Стратегическая важность карт контекстов

Отношения между ограниченными контекстами, технические и организационные, во многом более важны для групп разработчиков, работающих над проектом, чем сами ограниченные контексты. Информация, которую дают карты контекстов, помогает группам принимать важные стратегические решения, повышающие шансы проекта на успех. Карта контекстов — мощный артефакт, который поможет обнаружить вероятные неприятности на самых ранних этапах, а новым разработчикам — быстро влиться в работу. Карты контекстов могут также помочь вскрыть проблемы со взаимодействиями и рабочими процессами на предприятии.



Рис. 7.9. Карта контекстов с обозначениями типов связей между ограниченными контекстами

Сохранение целостности

Все разработчики в организации должны понимать карту контекстов. Группам не требуется понимать все внутренние процессы каждого ограниченного контекста; основное, что они должны знать об этих контекстах, — это экспортируемый ими прикладной программный интерфейс (API); отношения, которыми они связаны, и, что самое важное, их концептуальные модели. Владея всей этой информацией, группы смогут предотвратить размытие границ ответственности и гарантировать целостность всех контекстов.

Сохранение целостности помогает сосредоточить программный код на единственной модели. Это делает его более гибким, потому что любые изменения будут влиять только на один ограниченный контекст, а не распространяться по всей предметной модели, как круги по воде. Именно эта гибкость позволит вам быстро и уверенно изменять свой код и адаптировать его под изменяющиеся процессы и бизнес-логику.

Основа для планирования работ

Карта контекстов выявляет области хаоса и неразберихи и, что особенно важно, указывает, где находится смысловое ядро. Группы разработчиков могут использовать эту информацию для определения областей, куда в первую очередь должны быть направлены усилия:

- Области, в которых все зашло слишком далеко и эффект от усилий, направленных на дальнейшее совершенствование, не оправдывает затрат, можно изолировать и оставить.

- Области, не имеющие стратегической важности или имеющие низкую сложность, должны исключаться из процесса создания единого языка и для них не должны использоваться приемы проектирования на основе модели.
- Области, наиболее важные для успеха проекта или имеющие особенно сложную логику, являются первыми кандидатами для применения тактических приемов предметно-ориентированного проектирования и должны сохраняться изолированными от плохо спроектированных ограниченных контекстов для поддержания их целостности.

Понимание принадлежности и ответственности

Подотчетность и ответственность — другие нетехнические аспекты, которые могут влиять на успех проекта. Закрепление за группами разработчиков ответственности за подсистемы, с которыми нужно произвести интеграцию, поможет гарантировать своевременные изменения в соответствии с вашими ожиданиями. Создание карт контекстов — это непрерывное исследование и выяснение; возможно, вам не удастся нарисовать ясную карту контекстов с первой попытки, но сам процесс выяснения зон ответственности, определения границ и изучения потоков взаимодействий в ходе создания карты контекстов не менее важен, чем конечный артефакт.

Вскрытие запутанных областей в рабочих процессах на предприятии

Бизнес-процессы, протекающие между ограниченными контекстами и использующие их преимущества, часто остаются «бесхозными», без четко обозначенной ответственности за них и ясности в отношении их границ и способов интеграции. Карта контекстов, будучи ориентированной на нетехнические аспекты отношений, может вскрыть такие бизнес-процессы и потоки между системами и организационными подразделениями. Такие открытия часто весьма полезны для предприятия, потому что помогают лучше понять и усовершенствовать процессы, охватывающие отделы и подразделения. Это понимание можно использовать для снижения риска провала проекта путем устранения неоднозначности на ранних этапах и формулирования эффективных вопросов, помогающих добиться успеха проекта.

Часто серые области между контекстами, регламентирующие бизнес-процессы, также оказываются вне зоны чьей-либо ответственности, когда возникает необходимость внести изменения, и обнаруживаются позже, на следующих этапах жизненного цикла проекта.

Выявление нетехнических преград

Карты контекстов демонстрируют границы подразделений, вовлеченных в проект. Если ваша группа разработчиков владеет не всеми контекстами, над которыми ведется работа, вам придется уделить внимание вопросам расстановки при-

оритетов и координации своих действий с другими группами и специалистами других уровней управления. Понимание этих преград увеличивает шансы на успех проекта и позволяет заняться решением нетехнических проблем, таких как планирование выпусков новых версий, еще до того, как они начнут тормозить работу группы.

Аналогично, изменения, требующие интеграции со сторонними контекстами, могут потребовать организовать определенную среду тестирования и координировать действия с внешними группами или по крайней мере иметь доступ к экспериментальной среде и документации.

Способствование налаживанию общения

Когда диаграмма показывает вам, что ваш и некоторый другой ограниченные контексты связаны отношениями, вы должны понимать необходимость общения с группой, ответственной за этот другой контекст. Если вы также видите на диаграмме, что ваша группа отвечает за верхний контекст, вы должны понимать, что именно на вас будет лежать основная доля ответственности за принятие решений и, соответственно, часто вам первому придется инициировать коммуникацию.

Ускорение адаптации новых разработчиков

Приходилось ли вам начинать работу в новой компании и не понимать, какое место занимает ваша система среди других систем на предприятии? Чувствовали ли вы когда-нибудь неловкость, получая вопросы от специалистов в предметной области, из-за того, что озвученные ими проблемы затрагивают части системы, о существовании которых вы знаете лишь понаслышке? Краткая, но информативная карта контекстов, которая регулярно пересматривается и уточняется всеми членами группы, дает фантастическую возможность всем членам группы видеть и понимать всю картину в целом или хотя бы представлять, какие части системы они знают недостаточно хорошо. Если специалист придет с незнакомой вам проблемой, вы сможете обратиться к карте контекстов и посмотреть, к кому лучше обратиться за разъяснениями.

Ключевые идеи

- Карта контекстов отражает текущее положение вещей. Она обеспечивает целостное представление о технических методах интеграции и отношениях между ограниченными контекстами. Без этого есть риск нарушить целостность моделей, а ограниченные контексты могут быстро превратиться в большие «комья грязи» из-за размытия границ областей применимости моделей.
- Предохранительный слой обеспечивает изоляцию модели, взаимодействующей с другими контекстами. Он гарантирует нерушимость целостности, предоставляя средства преобразования представлений одного контекста в представления другого.

- Интеграция с применением шаблона «Общее ядро» хорошо подходит для контекстов, имеющих общие модели.
- Интеграция с применением шаблона «Служба с открытым протоколом» экспортирует внешний API, вместо того чтобы требовать от клиентов выполнять преобразование данных из одной модели в другую. Обычно при этом создается общий язык, предназначенный для работы с клиентами.
- Отношения между ограниченными контекстами можно выразить в терминах верхний/нижний. Верхний контекст оказывает влияние на нижний контекст.
- Сотрудничество между двумя группами, ориентированными на разные цели или работающими над разными проектами, часто называют отношением «заказчик — поставщик». Заказчики (находящиеся внизу) могут сотрудничать с поставщиками (находящимися сверху) для интеграции контекстов.
- Шаблон «Конформист» описывает отношение между верхней и нижней группами, когда верхняя группа не может пойти навстречу требованиям нижней группы. Часто такими верхними группами оказываются сторонние группы разработчиков.
- Шаблон «Партнерство» описывает отношение между двумя группами, имеющими общую цель и обычно работающими над одним проектом, но над разными контекстами.
- Шаблон «Отдельное существование» должен применяться, когда интеграция ограниченных контекстов оказывается слишком дорогим удовольствием и имеются другие, нетехнические методы интеграции.

8

Архитектура приложения

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Шаблоны архитектур приложений, защищающие целостность предметной модели
- Отличия между архитектурами приложений и ограниченных контекстов
- Роль и ответственность прикладных служб
- Как обеспечить поддержку разных клиентов приложения

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 8 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Целью философии DDD является преодоление проблем, возникающих при создании приложений со сложной предметной логикой, путем отделения предметных сложностей от сложностей технических. До сих пор в этой книге мы рассматривали лишь приемы, позволяющие группам разработчиков моделировать полезные концептуальные абстракции предметной области. В этой главе, напротив, рассматриваются шаблоны использования предметной модели в контексте приложения, с учетом вопросов, связанных с организацией хранения данных и их представлением, а также технических требований.

Архитектура приложения

Разработка программного обеспечения с применением принципов DDD не требует использовать какую-то определенную архитектуру организации приложений. Но какую бы архитектуру вы ни выбрали, она должна поддерживать изоляцию предметной логики.

Разделение задач, решаемых приложением

Чтобы избежать превращения программного кода в «большой ком грязи», утраты его целостности и, в конечном счете, ценности предметной модели, важно, чтобы структура приложения поддерживала отделение технических сложностей от сложностей предметной области. Представление, хранение и предметная логика в приложениях изменяются с разной скоростью и по разным причинам; архитектура, способствующая разделению этих задач, обеспечивает возможность внесения изменений без нежелательных эффектов на несвязанные между собой области.

Отделение от сложностей предметной области

Помимо разделения задач, архитектура приложения должна отделять сложности предметной области, скрывая низкоуровневые предметные детали за высокоуровневым интерфейсом. Абстрагирование на высоком уровне предотвращает изменение предметной логики под влиянием изменений на уровне представления и наоборот, так как клиенты взаимодействуют с приложением посредством прикладных служб, играющих роль интерфейса, и не имеют прямого доступа к предметным объектам.

Многоуровневая архитектура

Для поддержки разделения задач можно определить в приложении разные уровни, решающие разные задачи, как показано на рис. 8.1. В своей книге «Patterns of Enterprise Application Architecture»¹ Фаулер предложил обобщенную многоуровневую архитектуру. Однако существует множество других архитектур, поддерживающих разделение задач путем деления приложения на области, такие как «Чистая архитектура дядюшки Боба» («Uncle Bob's Clean Architecture»)², «Гексагональная архитектура» («Hexagonal Architecture», также известная как «Архитектура портов и адаптеров» — «Ports and Adapters Architecture») и «Луковая архитектура» («Onion Architecture»).

В отличие от типичных многоуровневых архитектур, центром архитектуры, изображенной на рис. 8.1, является предметный уровень (слой), включающий всю предметную логику. Предметный уровень окружен прикладным уровнем, за высокоуровневым прикладным интерфейсом (API) которого скрываются низкоуровневые детали предметной области. Уровни предметной и прикладной логики изолированы и защищены от влияния любых сложностей, связанных с взаимодействием с клиентами, использованием разных фреймворков и инфраструктурными задачами.

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: ООО «И. Д. Вильямс», 2009. (Ранее выходила в том же издательстве под названием «Архитектура корпоративных программных приложений»). — *Примеч. пер.*

² Псевдоним «Uncle Bob» («дядюшка Боб») принадлежит Роберту Мартину (Robert C. Martin), эксперту в области проектирования и разработки программного обеспечения. — *Примеч. пер.*

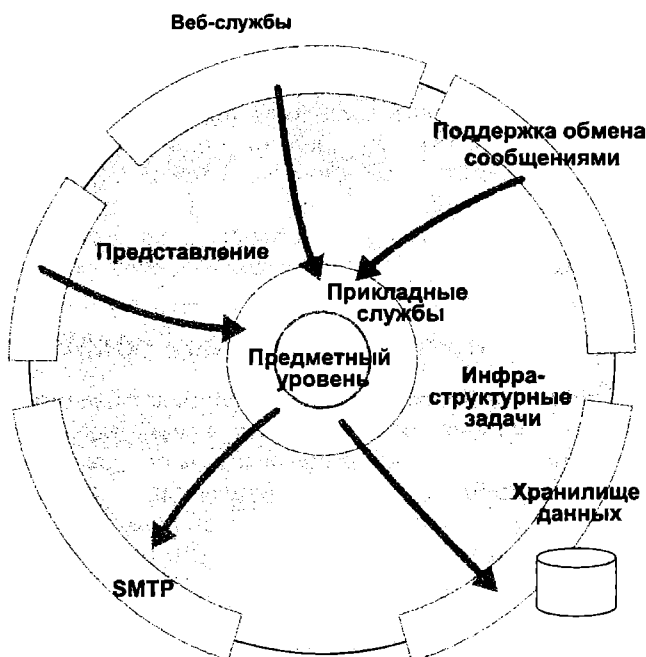


Рис. 8.1. Многоуровневая архитектура

Инверсия зависимостей

В целях обеспечения разделения задач предметный и прикладной уровни в центре архитектуры не должны зависеть ни от каких других уровней. Все зависимости обращены внутрь, то есть предметный уровень, находящийся в сердце приложения, вообще ни от чего не зависит, что позволяет его разработчикам сосредоточиться исключительно на предметных задачах. Прикладной уровень зависит только от предметного уровня; он организует обработку сценариев использования, обращаясь за решением к предметному слою.

Безусловно, состояния предметных объектов должны сохраняться в некотором хранилище данных. Чтобы добиться этого и избежать образования тесной связи предметного уровня с техническим кодом, прикладной уровень определяет интерфейс, позволяющий извлекать и сохранять предметные объекты. Этот интерфейс пишется с позиции прикладного уровня, в стиле, независимом от конкретных фреймворков или терминологии. Затем под эти интерфейсы реализуется и адаптируется уровень инфраструктурных задач, обеспечивающий поддержку, необходимую нижним уровням, без образования зависимостей. Управление транзакциями и сквозные задачи, например обеспечение безопасности и журналирование, реализуются таким же способом. На рис. 8.2 показаны направления зависимостей и направления интерфейсов, описывающие отношения между прикладным и техническими уровнями.

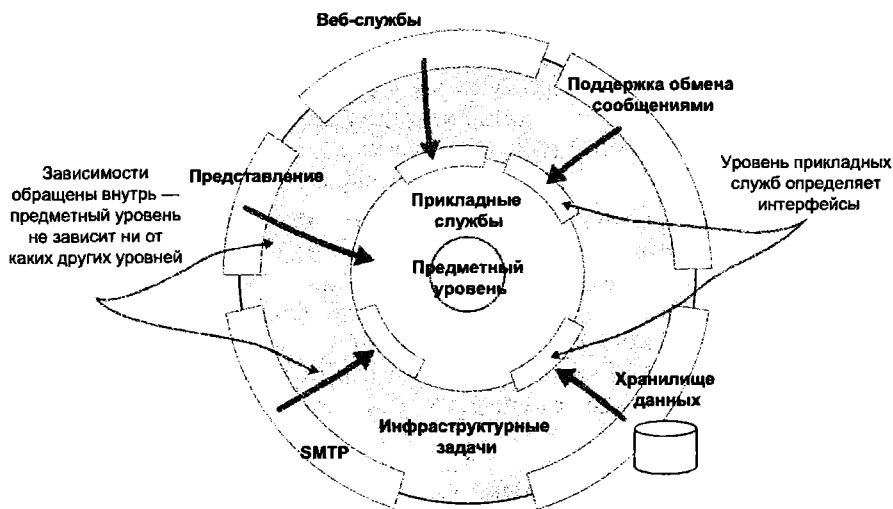


Рис. 8.2. Инверсия зависимостей в многоуровневой архитектуре

Предметный уровень

Как обсуждалось в главе 4 «Проектирование на основе модели», предметная модель представляет концептуальную абстракцию предметной области, созданную для удовлетворения потребностей предприятия. Предметный уровень, содержащий абстрактную модель, не зависит ни от чего другого и не предъявляет никаких требований к технической реализации обслуживаемых клиентов и хранилищ данных, где сохраняются предметные объекты.

Уровень прикладных служб

Уровень прикладных служб (или прикладной уровень) представляет сценарии использования и поведение приложения. Сценарии использования реализуются в виде прикладных служб, содержащих прикладную логику и координирующих выполнение сценариев с привлечением предметного и инфраструктурного уровней. Прикладные службы действуют на более высоком уровне абстракции, нежели предметные объекты, и экспортируют высокоуровневый интерфейс, за которым скрываются детали реализации предметного уровня, то есть они сообщают, что делает система, а не как она это делает. Скрывая сложность предметной области за фасадом, вы получаете возможность развивать предметную модель, гарантируя, что изменения в ней не затронут клиентов.

Уровень прикладных служб является клиентом предметного уровня; однако в своей работе он зависит также и от внешних уровней. Эти зависимости являются инвертированными, потому что контракты для требуемых интерфейсов определяются прикладным уровнем. Внешние ресурсы должны адаптироваться под определенные интерфейсы, чтобы избежать тесной зависимости прикладного уровня от конкретных технологий.

Извлечение предметных объектов из хранилища, передача им задания для выполнения и последующее сохранение измененного состояния объектов возлагаются на уровень прикладных служб. Уровень прикладных служб также отвечает за отправку извещений другим системам, когда в предметном уровне происходят важные события. Все эти интерфейсы к внешним ресурсам определяются внутри уровня прикладных служб, но реализуются в инфраструктурном уровне.

Уровень прикладных служб позволяет обеспечить поддержку самых разных клиентов без посягательств на целостность предметного уровня. Новые клиенты должны принимать данные в формате, определяемом контрактом приложения — его API. Также они обязаны передавать данные прикладным службам в формате, который им понятен. Таким образом, прикладной уровень можно рассматривать как своеобразный предохранительный слой, гарантирующий чистоту предметного уровня и его независимость от внешних технических деталей.

Инфраструктурные уровни

Инфраструктурные уровни приложения реализуют решение технических задач, обеспечивающих его работу. Если прикладной и предметный уровни определяют поведение приложения и предметную логику соответственно, то инфраструктурные уровни отвечают за решение исключительно технических задач, таких как взаимодействие с человеком посредством пользовательского интерфейса или с другими приложениями посредством комплекса веб-служб или конечных точек сообщений. Инфраструктурные уровни также отвечают за техническую реализацию хранения информации о состоянии предметных объектов.

Кроме того, инфраструктурные уровни могут реализовать поддержку журналирования, безопасности, рассылки извещений и интеграцию с другими ограниченными контекстами и приложениями. Все это технические детали, технические задачи, которые не должны оказывать прямого влияния на сценарии использования приложения и работу его предметной логики.

Взаимодействия между уровнями

При организации взаимодействий между уровнями, для предотвращения проникновения деталей предметной модели во внешний мир, следует сразу отказаться от прямой передачи предметных объектов через границы. По тем же причинам не следует передавать «сырые», незапрашиваемые данные или ввод пользователя внутрь предметного уровня. Для этой цели лучше использовать объекты переноса данных (Data Transfer Object, DTO), модели представления и объекты прикладных событий.

Чтобы избежать образования тесных связей между уровнями, верхние уровни должны взаимодействовать с нижними, преобразуя информацию в сообщения, которые нижние уровни готовы принимать. Это также поможет изолировать нижние уровни и обеспечить их независимость от верхних уровней. На рис. 8.3 показано, как можно организовать взаимодействия между уровнями и как осуществлять преобразование данных для защиты целостности предметной модели.



Рис. 8.3. Предметные объекты недоступны клиентам приложения

Тестирование в изоляции

Разделение задач в приложении и гарантия независимости предметной логики от любого технического кода, реализующего, например, уровень представления или хранение данных, позволяют выполнять тестирование предметной и прикладной логики по отдельности, независимо от инфраструктурного каркаса.

Как показано на рис. 8.4, для проверки логики работы предметного уровня можно использовать приемы модульного тестирования. Вы можете использовать фиктивные объекты и «заглушки» для передачи прикладному уровню фиктивной реализации требуемых ему зависимостей, чтобы подтвердить правильность координации взаимодействий с предметным уровнем и внешними ресурсами.

Не используйте схему данных, общую для всех ограниченных контекстов

Кроме разделения задач в программном коде приложения, архитектура должна предусматривать раздельное хранение состояний предметных объектов и других необходимых данных. На рис. 8.5 изображены приложения, интегрированные путем использования общей базы данных и общей схемы.

Этот метод интеграции действительно прост в реализации, но он может размывать границы моделей и действовать как катализатор превращения программного кода в «большой ком грязи». Наличие общих данных позволяет клиентам легко проникать через границы ограниченных контекстов и воздействовать на состояния предметных объектов, в обход защиты предметной логики. Также есть вероят-

ность неправильной интерпретации логики и схемы данных и внесения изменений, делающих эти данные недействительными.

Как показано на рис. 8.6, следует отдавать предпочтение базам данных, отдельным для приложений или ограниченных контекстов. По аналогии с тем, как возводятся границы контекстов в предметной модели, такие же границы нужно возводить и в модели хранения информации. Это поможет заставить клиентов произвести интеграцию с приложением через четко определенный уровень прикладных служб, защитить целостность модели и гарантировать достоверность данных.



Рис. 8.4. Тестирование уровней в изоляции

Сравнение архитектур приложений и ограниченных контекстов

Приложения могут состоять из нескольких ограниченных контекстов. Для организации приложений и ограниченных контекстов используются разные архитектуры. Приложение, включающее два или более ограниченных контекста, может использовать один архитектурный стиль для реализации пользовательского интерфейса и другие — для каждого из ограниченных контекстов. На рис. 8.7 изображено приложение, включающее три ограниченных контекста; здесь уровень представления имеет собственный прикладной уровень, чтобы упростить координацию действий с ограниченными контекстами.

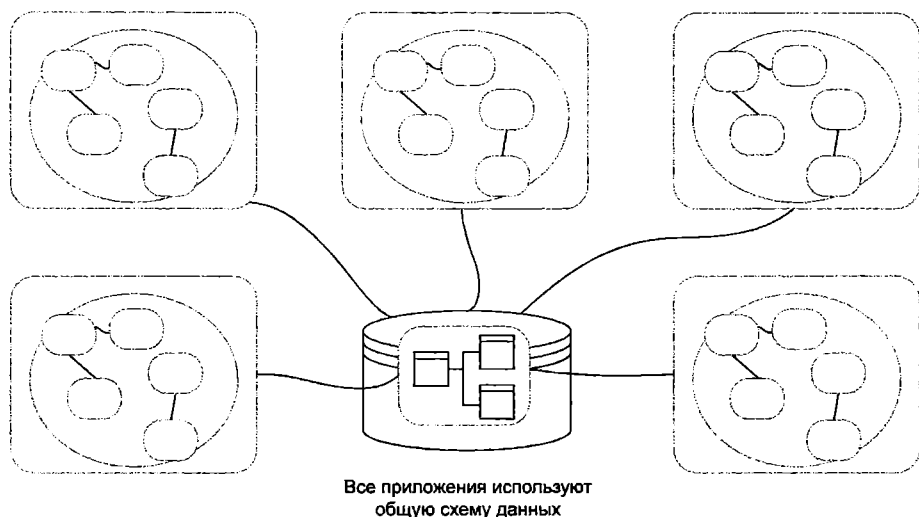


Рис. 8.5. Ограниченные контексты интегрированы посредством общей схемы данных

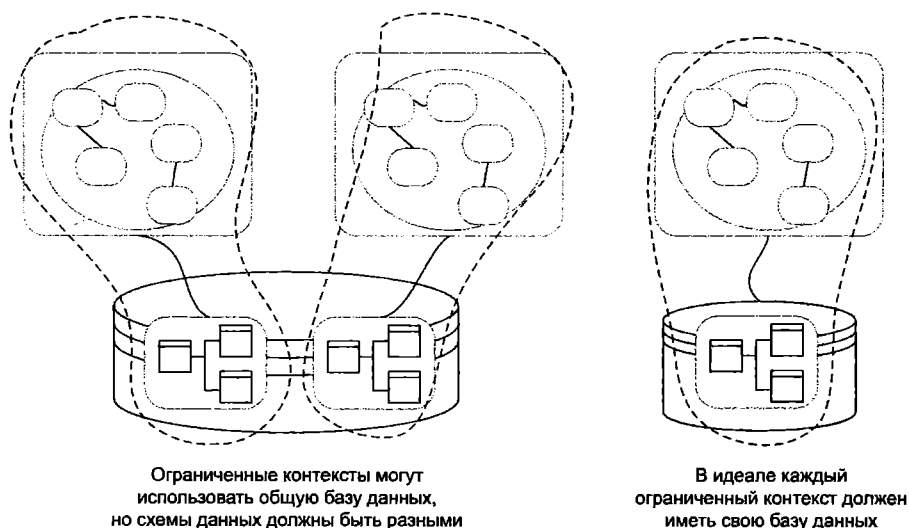


Рис. 8.6. Ограниченные контексты с отдельными схемами данных

Тем не менее некоторые полагают, что ограниченный контекст должен включать в себя также уровень представления. Бизнес-компонент Уди Дахана, например, возлагает на ограниченный контекст ответственность за управление собственной областью пользовательского интерфейса. Эту архитектуру можно увидеть на рис. 8.8.

В этом архитектурном стиле обязанность организации взаимодействий и распределения идентификаторов корреляции принимает на себя инфраструктурный слой.

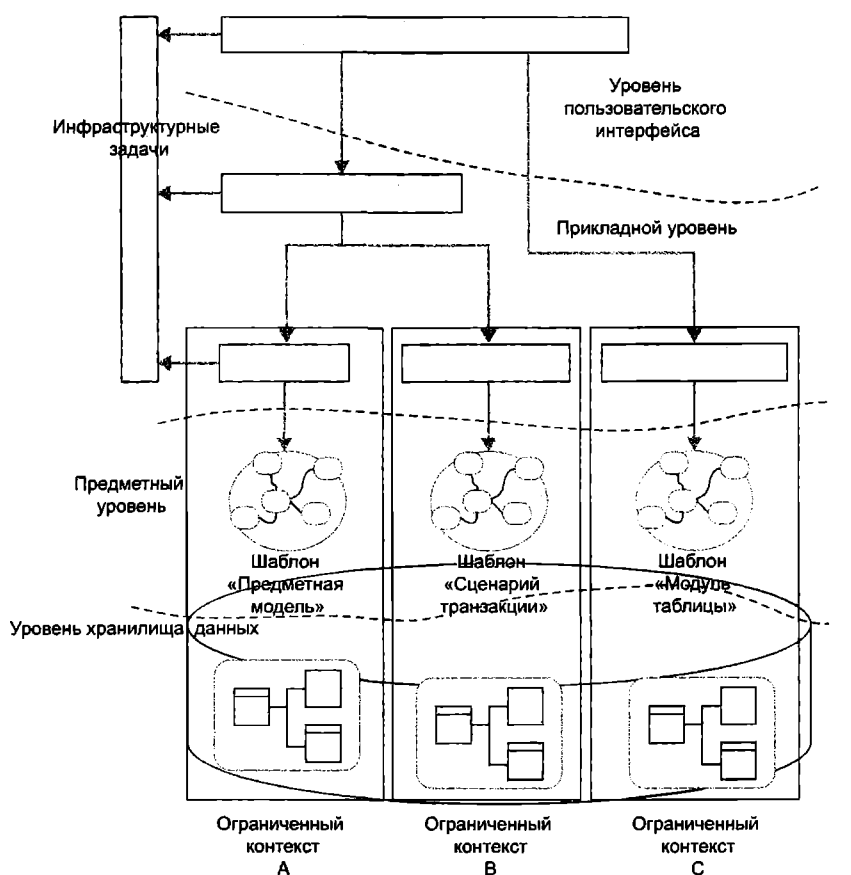


Рис. 8.7. Ограниченные контексты интегрируются посредством отдельного прикладного уровня

Ограниченные контексты, составляющие одно приложение, не должны следовать какому-то одному архитектурному стилю или использовать одно хранилище данных, но внутри ограниченного контекста желательно придерживаться единого метода представления предметной логики.

Прикладные службы

Уровень прикладных служб (application service layer), который классифицируется как уровень служб (service layer) в книге Фаулера «Patterns of Enterprise Application Architecture», можно использовать для определения границ предметной модели и рассматривать как реализацию понятия ограниченного контекста, изолирующую и защищающую целостность этой модели.

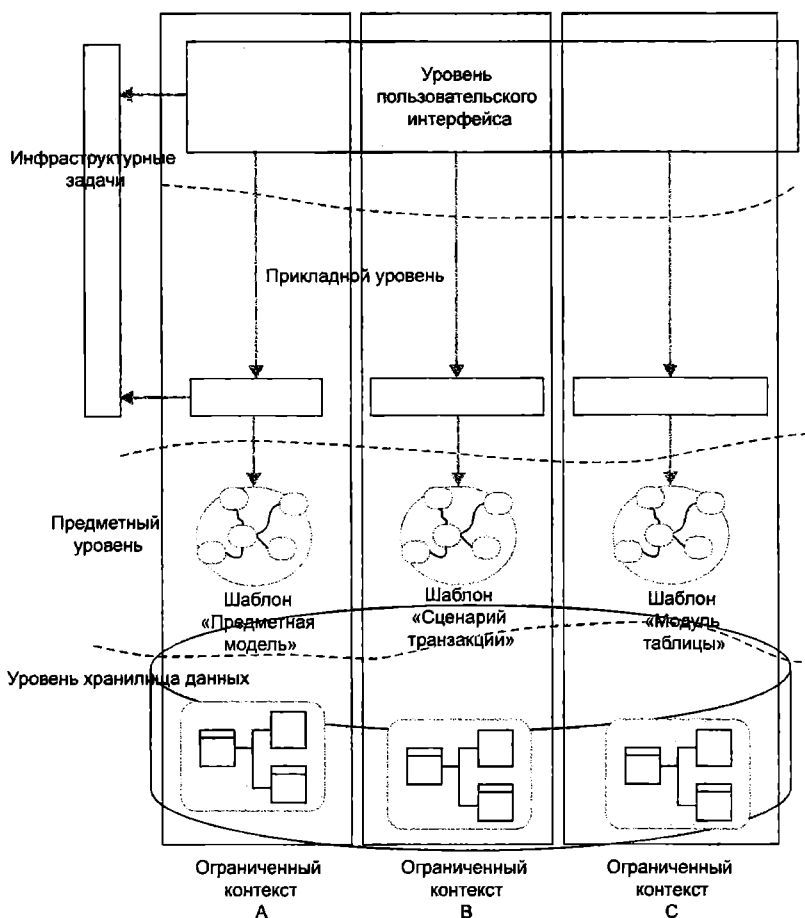


Рис. 8.8. Уровень представления составлен из ограниченных контекстов

Как отмечалось выше в этой главе, назначение уровня прикладных служб заключается в экспортировании приложению функциональных возможностей предметной модели и сокрытии деталей за высокоуровневым интерфейсом. Функциональные возможности определяются сценариями использования, которым должна соответствовать система. Прикладные службы выполняют эти сценарии, координируя работу предметной логики. Им приходится иметь дело с техническими тонкостями, такими как обработка ввода и формирование отчетной информации о состоянии предметной модели, а также управлять транзакциями, осуществлять журналирование и выполнять операции с хранилищем данных.

Прикладные службы содержат только прикладную логику. Эта логика отвечает за решение таких задач, как безопасность, управление транзакциями и взаимодействие с другими техническими возможностями, такими как электронная почта и веб-службы. Они являются клиентами предметного уровня и всю работу делегируют ему. Никакая предметная логика не должна выполняться прикладными служ-

бами — прикладные службы должны быть максимально простыми и написанными в процедурном стиле. Прикладной уровень не зависит ни от каких фреймворков или технологий, используемых прикладными службами, таких как фреймворки пользовательского интерфейса или вспомогательные фреймворки. Тем не менее он должен определять интерфейсы, которые зависят от представления предметных объектов, и управлять задачами, не связанными с предметной областью.

Прикладная и предметная логика

Прикладная логика содержит этапы, необходимые для выполнения сценариев использования. Этапы могут включать извлечение предметных объектов из базы данных и их преобразование, отображение ввода пользователя в объекты, которые принимает предметный уровень, и делегирование принятия решения предметным объектам или их коллекциям. В число этапов могут также входить обращения к инфраструктурным службам, таким как рассылка извещений об изменениях в состоянии предметной модели посредством службы сообщений или веб-вызовов, авторизация и журналирование.

Основное назначение прикладной логики — координация и организация посредством делегирования задач предметному и инфраструктурному уровню. Прикладные службы вообще не делают никакой полезной работы, но они знают, к кому обратиться для решения задачи. Предметная логика, напротив, опирается только на предметные правила, понятия, информацию и процессы. Предметная логика свободна от технических деталей, включая работу с хранилищем данных.

В качестве примера рассмотрим рис. 8.9, где изображена модель сценария применения к корзине рекламного купона, дающего право на скидку. Уровень представления, реализованный на основе фреймворка ASP.NET MVC, преобразует HTTP-запрос в форму, понятную уровню прикладных служб, и вызывает метод службы. Прикладная служба обращается к уровню хранилища данных, чтобы получить объект купона. Затем проверяет действительность купона. Если купон недействителен, возвращает соответствующий результат. Если купон действителен, она снова обращается к уровню хранилища данных, чтобы получить объект корзины, и передает его объекту купона для вычисления скидки. Изменения, полученные после применения скидки к предметному объекту корзины, сохраняются в хранилище, затем посылается событие, извещающее о погашении купона.

Определение и экспортирование функций

В связи с тем что прикладные службы экспортируют функциональные возможности, они не должны адаптироваться под требования новых клиентов. Вместо этого новые клиенты, такие как уровни представления, должны выполнять требования контрактов, экспортируемых службами. Иными словами, функции системы не должны изменяться под требования клиентов. Изменять их следует только при изменении сценариев использования. Сценарии использования, экспортируемые прикладными службами, изменяются реже и по причинам, отличным от изменений предметной логики, выполняющей их. Это позволяет защитить клиентов, пользующихся службами, от частых изменений в предметной логике.

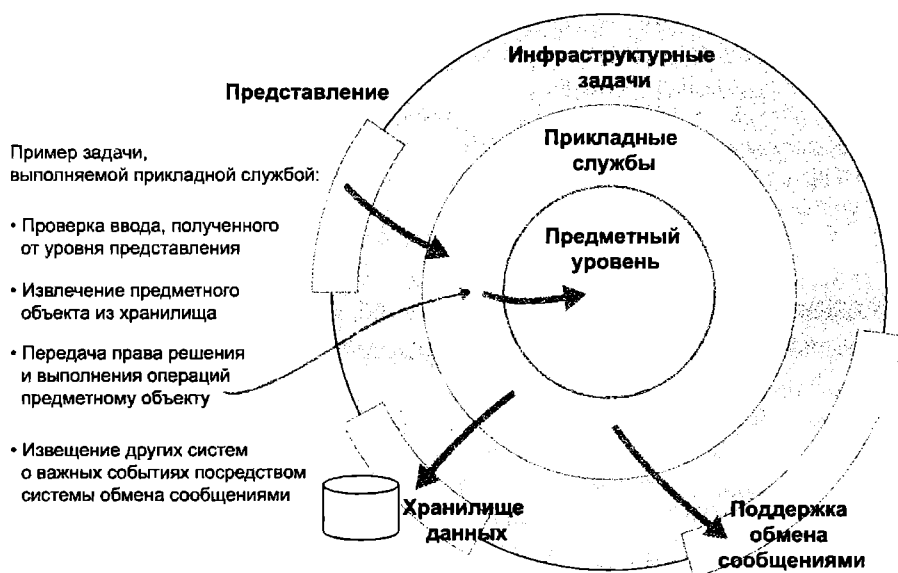


Рис. 8.9. Прикладная и предметная логика

Возьмем, к примеру, сценарий оценки риска столкновения с мошенническим заказом. Система экспортирует функцию, которая получает информацию о заказе и возвращает оценку, вычисленную предметной логикой. С течением времени предметная логика может измениться, но прикладная служба, реализующая сценарий оценки риска, почти наверняка останется прежней и изменится, только если изменившийся контракт потребует передачи дополнительной информации.

Пользователи могут не знать всех сложностей предметного уровня; для них большее значение имеют бизнес-задачи, за которые отвечает прикладной уровень. Даже не являясь специалистами в предметной области, пользователи должны понимать их.

Координация выполнения сценариев использования

Если предметная модель имеет объектно-ориентированную природу, то прикладные службы по своей сути являются процедурами, так как в первую очередь они ориентированы на управление выполнением задач, а не на моделирование предметной логики и понятий. Прикладные службы можно сравнить с методами действий контроллера ASP.NET MVC. Действия контроллера содержат логику, управляющую взаимодействиями с пользовательским интерфейсом, также и прикладные службы содержат логику, управляющую выполнением сценариев использования и координирующую взаимодействия со службами и объектами в предметном слое. И те и другие — действия контроллеров и прикладные службы — не имеют собственного состояния и являются процедурами по своей природе. Единственное исключение: действия контроллеров и прикладные службы могут сохранять состояние, но только если это информация о цикле взаимодействий с клиентом или о ходе выполнения деловой задачи.

Сама логика работы прикладных служб также напоминает логику действий контроллера — основной ее задачей является преобразование входных данных и отображение их в выходные. Действия контроллеров отображают HTTP-переменные в объекты, которые могут обрабатывать веб-службы, и отображают ответы веб-служб в модели уровня представления. Прикладные службы действуют аналогично, отображая запросы в структуры, понятные предметным объектам, и возвращают модели представления, которые скрывают истинную организацию предметных объектов и понятны пользовательскому интерфейсу.

Прикладные службы представляют сценарии использования, а не CRUD-операции

Проектирование на основе функциональности (Behavior-Driven Design, BDD) поможет понять поведение приложения. Определив функции с применением подходов BDD, можно использовать язык, выражающий спецификации BDD, для именования сценариев использования прикладных служб. Это напоминает использование единого языка (Ubiquitous Language, UL) предметной области в программном коде предметного уровня. Прикладные службы — это не просто набор методов создания, чтения, изменения и удаления; они должны сообщать пользователям свои цели и раскрывать возможности системы. Примеры можно увидеть в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования», где также представлены шаблоны реализации уровня прикладных служб.

Предметный уровень как детали реализации

Прикладные службы — мощный инструмент, который с успехом может решать любые прикладные задачи, от очень сложных, с запутанной логикой, до простых, обеспечивающих доступ к хранимым данным. Прикладные службы отделяют клиентов от логики и позволяют развивать предметный уровень, не опасаясь распространения побочных эффектов через слои.

Методы прикладных служб могут сообщать вам, нужна ли вообще предметная модель. Если обнаружится, что все сценарии использования сводятся к простому изменению, добавлению или удалению данных, это явный признак того, что в предметной области нет никакой действительной логики. В таком случае ее реализацию можно упростить, применив шаблон «Сценарий транзакции» или «Обертка данных» (data wrapper), как описывалось в главе 5 «Шаблоны реализации предметной модели», вместо создания полноценной предметной модели. Однако если ваша система имеет богатое многообразие прикладных служб и функций, это может служить признаком необходимости применения шаблона «Предметная модель» для реализации предметного уровня.

Отчеты о состоянии предметной модели

Помимо координации выполнения деловых задач, уровень прикладных служб должен предоставлять информацию о состоянии предметных объектов в форме отчетов. Чтобы не раскрывать детали работы предметной модели внешнему миру,

прикладные службы должны преобразовывать предметные объекты в модели представления, сообщаящие информацию о состоянии предметной модели, не раскрывая ее структуру. Увидеть это преобразование можно на рис. 8.10.

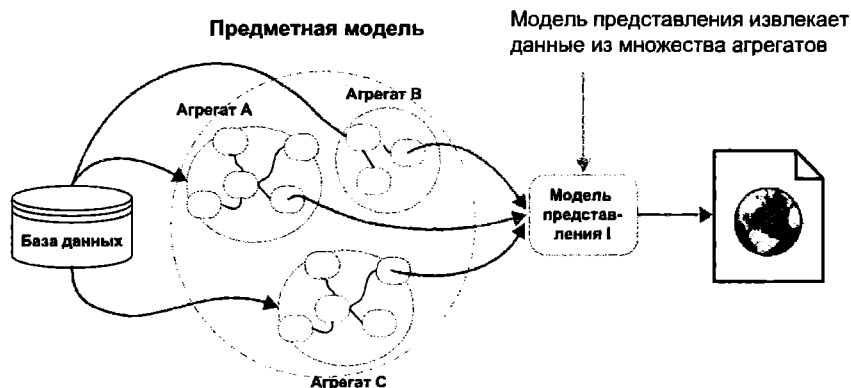


Рис. 8.10. Модель представления отображается во многие предметные объекты

Модели чтения и транзакционные модели

Иногда в пользовательском интерфейсе требуется отобразить информацию, собранную из множества предметных объектов. Проведение процедуры извлечения всех предметных объектов только для того, чтобы предоставить подмножество информации для отображения, может оказаться дорогой и неэффективной. В таких случаях часто предпочтительнее предоставить для уровня прикладных служб некоторое представление состояния предметной области на основе исходных данных, как показано на рис. 8.11. При таком подходе можно конструировать эффективные представления без необходимости создавать большие иерархии предметных объектов.

Однако предоставление функций чтения/записи для одной и той же концептуальной модели имеет свои недостатки. Транзакционная модель обеспечивает хранение логики в предметных объектах, а простого состояния — в хранилище данных. Для поддержки потребностей отчетов и транзакций представлениям может понадобиться дополнительная информация, что, в свою очередь, может повлечь изменения в структуре предметных объектов. Чтобы предотвратить изменения предметной модели в угоду требованиям уровня представления, данные для представления можно хранить отдельно, выполнив оптимизацию схемы их хранения под запросы. Для этого можно организовать сохранение изменений в предметной модели и использовать их как основу для нужд составления отчетов.

На рис. 8.12 показано, как транзакционная модель обрабатывает запросы на запись от клиентов и затем генерирует события, которые сохраняются для последующих запросов. События и их данные можно сохранять в той же базе данных или в совершенно другом хранилище. Этот шаблон называется «Разделение ответственности команд и запросов» (Command Query Responsibility Segregation, CQRS) и подробно описывается в главе 24 «CQRS: архитектура ограниченного

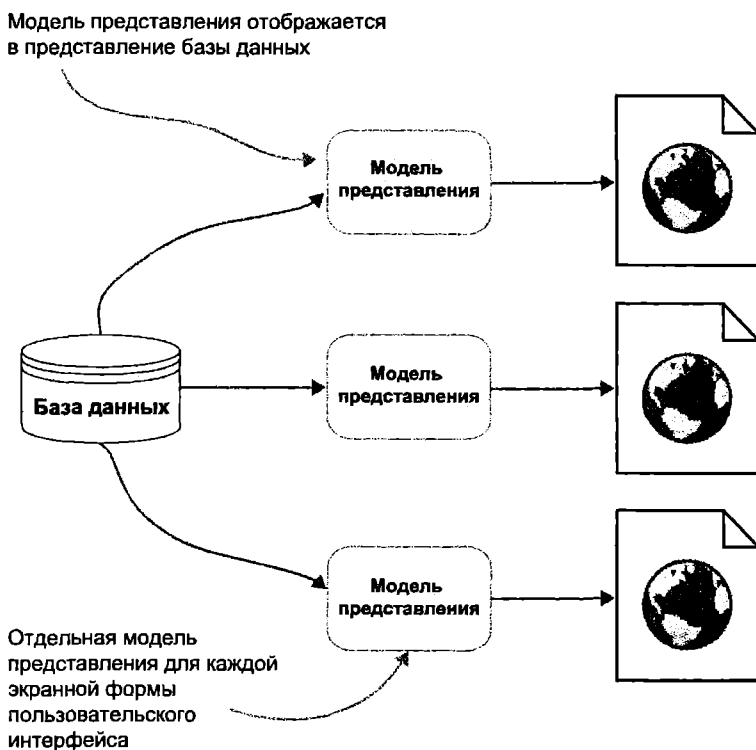


Рис. 8.11. Модели представления запрашивают данные непосредственно из хранилища

контекста». Дополнительные шаблоны реализации отчетов о состоянии предметной модели представлены в главе 26 «Запросы: предметная отчетность».

Клиенты приложения

Роль клиентов уровня прикладных служб заключается в экспортировании функциональных возможностей системы. Многие приложения имеют уровень представления в виде пользовательского интерфейса, предоставляющего пользователям доступ к функциям системы. Другие приложения, напротив, экспортируют свою функциональность посредством веб-служб или служб RESTful. Независимо от типа клиентского приложения, служба не должна зависеть от того, кто пользуется ее функциональностью. Прикладные службы не должны соответствовать потребностям клиентов — они должны экспортировать сценарии использования приложения и вынуждать клиентов адаптироваться под предоставляемый прикладной программный интерфейс.

В целом можно построить систему без уровня прикладных служб, переложив на клиентов выполнение всех задач, за которые должен отвечать прикладной уровень. Однако при создании определенного множества прикладных служб явно модели-



Рис. 8.12. Представление хранится отдельно от транзакционного хранилища



Рис. 8.13. Различные клиенты приложения

руются сценарии использования, которые затем отделяются от любых потребностей представления. Такие прикладные службы помогают сосредоточиться на поведении системы и позволяют отделить предметную логику от других функций приложения. На рис. 8.13 показано, как множество клиентов пользуются функциями приложения посредством уровня прикладных служб. Здесь также показано, как сами прикладные службы могут пользоваться внешними контекстами и сторонними службами.

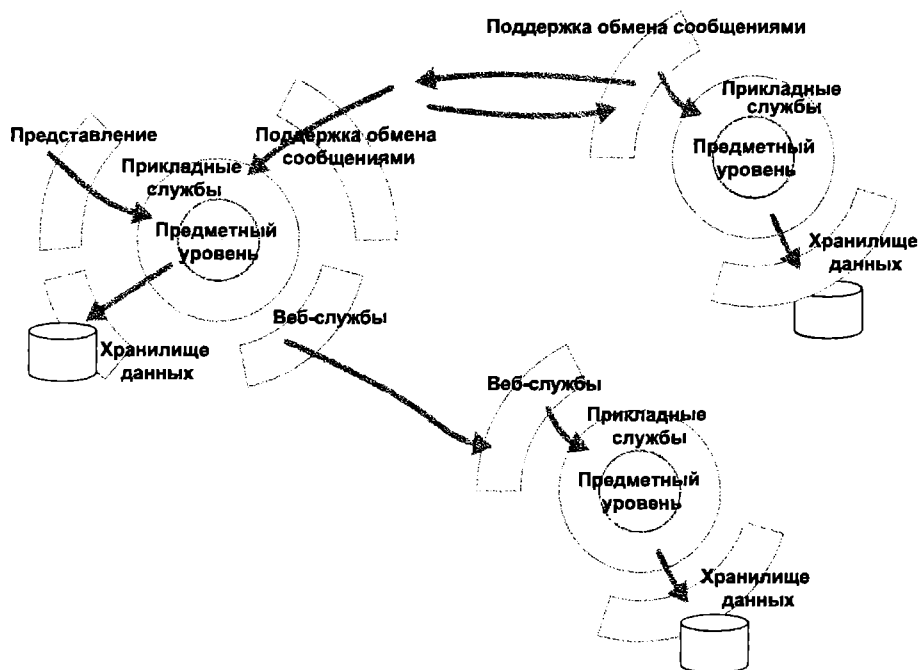


Рис. 8.14. Система, состоящая из множества ограниченных контекстов

Ограниченные контексты могут образовывать огромные системы, взаимодействуя друг с другом через техническую инфраструктуру. На рис. 8.14 показаны разные клиенты, в результате совместной работы которых формируется большая система. Методы интеграции ограниченных контекстов будут показаны во второй части книги, а потребности интерфейсов пользователя — в четвертой части.

Иногда бизнес-процессы охватывают несколько ограниченных контекстов. В таких случаях обычно используются диспетчеры процессов, координирующие выполнение бизнес-задач. На рис. 8.15 представлен диспетчер процесса, содержащий логику координации больших процессов. Подобно прикладным службам, диспетчеры процессов не будут иметь информации о своем состоянии, кроме информации, необходимой для отслеживания хода решения задачи, а всю фактическую работу будут делегировать обратно приложениям. Этот шаблон мы рассмотрим в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования».

Ключевые идеи

- Философия DDD не требует использования какой-то определенной архитектуры — она допускает применение любой архитектуры, отделяющей технические задачи от предметных задач.

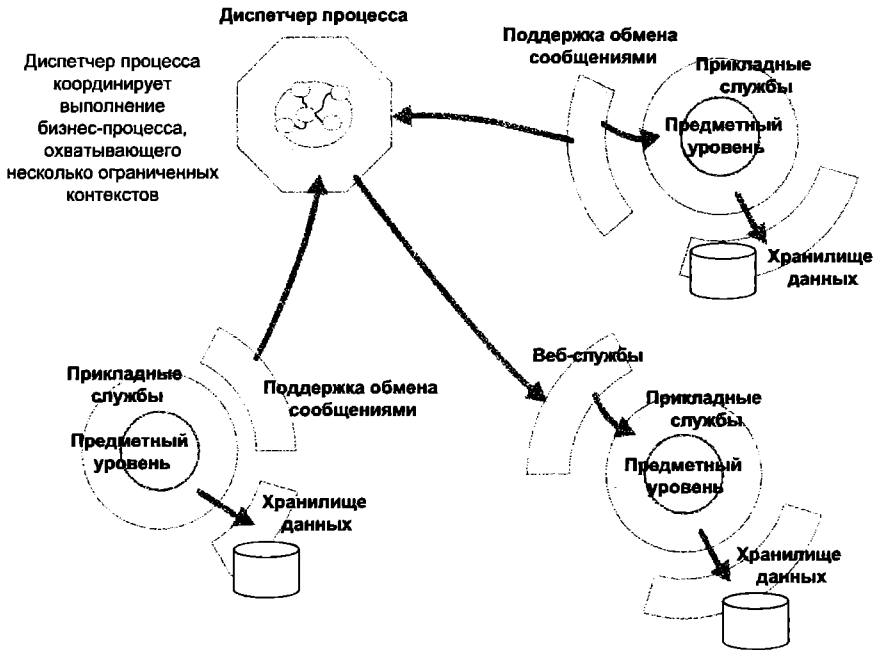


Рис. 8.15. Диспетчер процесса

- Разделяйте задачи приложения и отделяйте сложности предметной области от технических сложностей путем многоуровневой организации приложения.
- Внешние уровни зависят от внутренних. Внутренние уровни определяют интерфейсы, которые должны реализовать и использовать внешние уровни. Такая форма инверсии зависимости защищает целостность предметного и прикладного уровней.
- Предметный уровень — это основа приложения. Он должен быть изолирован от технических сложностей с помощью прикладного уровня.
- Прикладные службы экспортируют функциональные возможности системы путем абстрагирования предметной логики на более высоком уровне.
- Прикладные службы создаются на основе сценариев использования и являются клиентами предметного уровня. Выполнение сценариев использования они делегируют предметному уровню.
- Прикладные службы не должны зависеть от клиентов, которые используют их функциональность. Именно клиенты должны адаптироваться под программный интерфейс приложения. Именно такой подход позволит обеспечить поддержку самых разных клиентов.
- Уровень прикладных служб — это конкретная реализация границ ограниченного контекста.

9

Типичные проблемы команд, начинающих применять предметно-ориентированное проектирование

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Почему DDD это нечто большее, чем просто написание программного кода
- Как не переоценить важность тактических шаблонов
- Как неправильное применение DDD может превратить простую задачу в сложную и вызвать разочарование
- Почему стратегическое проектирование, сотрудничество и общение важнее, чем язык шаблонов DDD
- Как группы, не фокусирующиеся на смысловом ядре, тратят свои силы впустую
- Какие ловушки подстерегают разработчиков, применяющих DDD без привлечения специалистов в предметной области или без использования методологии итеративной разработки
- Как антишаблон помогает бороться за совершенство в DDD

Предметно-ориентированное проектирование — это философия, появившаяся из потребности сместить фокус внимания групп разработчиков, создающих сложные программные продукты. Это не фреймворк и не множество предопределенных шаблонов, которые можно применить к проекту. Не умаляя значимости действительно полезных фреймворков и шаблонов, DDD подчеркивает ценность сотрудничества, исследований и изучения предметной области разработчиков и специалистов для получения полезного и качественного программного обеспечения. Группы разработчиков должны объять предметную область, в которой работают, и выйти за рамки технической узости своего видения. Самым главным на пути к созданию великолепного программного обеспечения является понимание предметной области, и значимость этого понимания не должна оцениваться ниже технических знаний.

Группы, не знакомые с философией DDD или не до конца понимающие основные ее концепции, могут столкнуться с некоторыми проблемами в процессе ее применения в своих проектах. В данной главе мы расскажем о таких типичных проблемах и объясним, почему некоторые группы с трудом принимают эту философию.

Переоценка важности тактических шаблонов

DDD включает набор тактических шаблонов, помогающих в проектировании на основе модели и создании предметной модели. Если вы наберете фразу «DDD» в поиске Google, вы увидите, что важность этих строительных блоков часто переоценивается и нередко они ошибочно воспринимаются как наиболее важная часть DDD. Сам Эванс¹ часто корит себя за то, что не поместил в начало своей книги описание большего числа стратегических шаблонов DDD вместо тактических, потому что многие прекращают чтение после знакомства с этим набором строительных блоков.

Сосредоточенность на тактических шаблонах DDD представляет большую проблему: появляются группы разработчиков, концентрирующихся при создании программного кода только на технических шаблонах. Разработка кода при проектировании систем даже со сложной логикой никогда не была узким местом. Код — это результат совместной работы разработчиков и специалистов над моделированием предметной области. Код — это конечный результат сотрудничества и исследований. Цель разработчика состоит в том, чтобы решить задачу, а решить ее порой намного легче вместе со специалистами, вдали от клавиатуры. Наконец, действующий код — это результат изучения и освоения предметной области.

Использование одной архитектуры для всех ограниченных контекстов

Философия DDD не ограничивает выбор фреймворка, инструментов или архитектуры приложения. Разработчик не обязан использовать CQRS, события, службы RESTful, механизмы обмена сообщениями или средства объектно-реляционного отображения для применения принципов философии DDD. Единственное, на чем настаивает эта философия, — предметная модель должна отделяться от технического кода. Любая архитектура, отвечающая этому требованию, хорошо согласуется с DDD. Предметная логика и технический код изменяются с разной скоростью; как результат — выделение их в разные контексты обеспечивает простоту управления сложностями.

Архитектуры должны выбираться для ограниченных контекстов, а не для приложения. Для простого ограниченного контекста с несложной логикой можно использовать комбинацию многоуровневой архитектуры с шаблоном «Сценарий транзакции» для предметного уровня. В более сложных ситуациях можно применить архитектуру CQRS и шаблон «Предметная модель» для предметного уровня.

Идеализация тактических шаблонов

Группы, которые сконцентрированы на создании программного кода, сосредотачиваются на тактических шаблонах DDD. Они воспринимают эти шаблоны как

¹ Эрик Эванс, автор книги «Предметно-ориентированное проектирование. Структуризация сложных программных систем». — *Примеч. пер.*

библию, а не руководство, не понимая, что иногда правила стоит нарушать. Они впустую тратят массу усилий, придерживаясь шаблонов. Эту энергию было бы целесообразнее направить на понимание, зачем нужно писать данный программный продукт. Философия DDD учит искать ответы на вопросы: «Что нужно написать?», «Зачем это нужно написать?» и «Каким областям следует уделить особое внимание?». Как упоминалось прежде, тактические шаблоны DDD — это элементы, ставшие известными во многом благодаря появлению книги Эрика, в центре внимания которой находится все-таки стратегическая сторона DDD. Не так важно, как именно разработчики будут создавать предметные модели, важнее понять, что эти модели должны стоять во главе угла и развиваться в рамках ограниченных контекстов. Осознание того, что является наиболее важным для решения задачи и почему, намного важнее, чем ответ на вопрос: «Как реализовать это в коде?».

Ошибочное принятие строительных блоков за ценность DDD

Многие проекты DDD потерпели неудачу из-за того, что для их реализации были выбраны тактические шаблоны, а стратегическая сторона DDD и сотрудничество проигнорированы. Разработчики не выделяли времени на переработку знаний со специалистами в предметной области. Они не концентрировали усилия на создании предметной модели и невнимательно подходили к выбору абстракций. Они не создавали единого языка (UL). Использование только лишь языка тактических шаблонов DDD часто называют облегченной философией DDD (DDD lite). Следование облегченной философии не является заблуждением, но это не полная философия DDD. Многие ошибочно думают, что применяют философию DDD, отбрасывая самые важные ее аспекты: сотрудничество, единый язык и ограниченные контексты. Сосредоточенность только на шаблонах, помогающих проектированию на основе модели, не позволяет увидеть общую картину решения задачи.

Напротив, многие стратегические шаблоны DDD с успехом можно использовать в работе над средне- и крупномасштабными системами, независимо от их сложности. Фактически все стратегические шаблоны имеют множество положительных сторон, включая выявление необходимости в создании единого языка и использовании тактических шаблонов. Деление на подобласти может помочь разбить сложные предметные области на части для облегчения общения и определения наиболее важных сторон задачи. Карты контекстов вскрывают не только точки интеграции между разными контекстами, но и структуру взаимоотношений между группами разработчиков. Однако сложно оправдать тактические шаблоны, предписывающие применение шаблона «Предметная модель» к чему-то иному, кроме как к сложным или постоянно меняющимся областям.

Сосредоточенность на коде, а не на принципах DDD

На форумах разработчики программного обеспечения часто задают вопрос: «Можно ли увидеть пример приложения?». Вероятно, есть масса хороших решений, демонстрирующих конечный результат разработки с применением филосо-

фии DDD, но, фокусируясь только лишь на исследовании программного кода, вы не сможете увидеть главные преимущества DDD. Воплощение принципов DDD происходит на доске, за чашкой кофе и в коридорах, при встречах со специалистами в предметной области; оно проявляется в виде мелких изменений кода, когда неожиданно вскрываются новые предметные понятия, позволяющие глубже вникнуть в предметную область. Пример приложения не поможет увидеть, как протекало общение и сотрудничество между специалистами и разработчиками.

Программный код — это результат месяцев и месяцев упорного труда, но он представляет лишь последнюю итерацию. Сам код поменял множество обличий, прежде чем принял текущий вид. С течением времени код продолжит развиваться для поддержки изменяющихся потребностей предприятия; модель, используемая сегодня, в будущих итерациях может измениться до неузнаваемости.

Если рассматривать только решение, созданное с применением DDD в надежде сымитировать философию, многие идеи и приемы окажутся неопробованными и неоправданно большое значение будет придано строительным блокам в коде. Фактически, не имея глубоких познаний в предметной области, нельзя оценить всю выразительность предметной модели.

Философия DDD действительно предлагает множество эффективных приемов проектирования, шаблонов и строительных блоков, которые часто ошибочно принимают за ее суть. Рассматривайте все эти артефакты проектирования лишь как средства достижения конечной цели, используемые для представления принципиальной модели. Суть DDD находится глубже и заключается в сотрудничестве разработчиков со специалистами в предметной области с целью получить эффективную модель.

Недооценка истинной ценности DDD: сотрудничество, общение и контекст

Группа, слишком сосредоточенная на тактических шаблонах, упускает из виду суть DDD. Истинная ценность DDD заключается в создании общего языка, характерного для контекста и позволяющего разработчикам и специалистам эффективно сотрудничать над решением. Программный код — лишь побочный результат этого сотрудничества. Целью является устранение неоднозначностей в терминологии и легкость в общении. Эта цель должна быть достигнута еще до того, как вы приступите к созданию кода, тогда у вас появятся отличные шансы решить поставленные задачи. Только в том случае, когда разработка начинается с определения единого языка, уточнения контекста и налаживания сотрудничества, код получается хорошо организованным и однозначно связанным со смысловыми моделями предприятия.

Задачи решаются не только в коде, но также через сотрудничество, общение и совместные исследования со специалистами в предметной области. Уровень разработчиков должен оцениваться не по тому, как быстро они умеют «штамповать» код, а по тому, как они решают задачи.

Получение «большого кома грязи» из-за недопонимания важности контекста

Контекст, контекст и еще раз контекст. Это основная идея DDD. Контекст помогает организовать решения для больших предметных областей. Одна и та же модель не может решить все задачи. Для разных задач следует создавать разные модели. Создание моделей в пределах определенных границ контекстов поможет удерживать код в управляемом состоянии и избежать его превращения в «большой ком грязи». За представление границ контекстов отвечает карта контекстов. Группы, пренебрегающие контекстами и картами контекстов, не смогут произвести что-то ценное для предприятия, потому что постоянно будут бороться с неорганизованным беспорядком в программном коде.

Если группы разработчиков остаются в неведении о существовании других контекстов, вносимые ими изменения могут просачиваться в эти другие контексты. Без четкого знания границ модели увеличивается вероятность нарушения ее концептуальной целостности. Размытые границы между моделями или их отсутствие часто ведет к слиянию моделей, что довольно быстро может привести к превращению кода в «большой ком грязи». Карта контекстов жизненно необходима для понимания границ и поддержания целостности моделей.

Самой большой проблемой при изменении унаследованного кода с соблюдением технических требований является его организация. Код легко написать, но если не проявить должного внимания к его организации, такой код может стать очень тяжелым для чтения. Знание контекстов позволит вам изолировать несвязанные идеи, чтобы модели оставались ясными и специализированными. Рассматривайте это как применение принципа единственной ответственности (Single Responsibility Principle, SRP), только на архитектурном уровне. Код, простой в чтении и сопровождении, позволит группам разработчиков быстрее создавать новые ценности для предприятия, что и является сутью DDD.

Отличайте ситуации, когда специалисты используют одинаковые термины в разных контекстах. Если те же термины используются на предприятии в разных контекстах, легко попасть в ловушку, думая, что некоторые модели можно использовать повторно. Остерегайтесь неявных моделей, которые используются более чем в одном контексте. Более приемлемым вариантом будет явное определение моделей. Применяйте принцип «не повторяйся» (Don't Repeat Yourself, DRY) только к ограниченным контекстам, но не к системе в целом. Не бойтесь использовать те же идеи и названия в разных контекстах. Самое важное, что должны понимать разработчики, — это необходимость защиты своих границ.

Появление неоднозначности и недопонимания из-за неудач при создании единого языка

Эффективное общение — залог понимания и преодоления проблем в предметной области. Без регулярного общения невозможно наладить всестороннее сотрудничество между разработчиками и специалистами в предметной области. Разработчики, не оценившие важность общего языка, почти наверняка начнут использовать

технические абстракции и конструировать модель с использованием собственного общего технического языка. Когда таким разработчикам понадобится обратиться к специалистам за помощью или представить модель для утверждения, им придется переводить техническое описание на язык, понятный специалистам. В лучшем случае такой перевод будет тормозить процесс разработки; в худшем — дело может закончиться тем, что разработчики будут строить модели на основе идей, не играющих важной роли или непонятных специалистам в предметной области.

Без общего языка невозможно создать общую модель. Совместное исследование проблемы помогает увидеть скрытые понятия и совместно выработать ее решение. Процесс создания языка — это прямой результат сотрудничества между разработчиками и специалистами в предметной области. Возможность легко решать проблемы и понимание предметной области — вот основные выгоды этого процесса.

Без единого языка очень сложно выделить контексты, потому что ограниченный контекст в первую очередь определяется применимостью языка. Модели, созданные вне контекста и без использования единого языка, быстро превращаются в «большой ком грязи», так как концепции с одинаковыми названиями моделируются все вместе.

Формирование языка может оказывать большое влияние на бизнес в целом и разработку программного продукта. Этот процесс, так же как и язык шаблонов, помогает явно определить общие понятия и устранить неоднозначности при обсуждении сложной предметной логики и организации предприятия.

При наличии единого языка специалисты в предметной области смогут предлагать решения проблем, возникающих при реализации предметной модели, наравне с самими разработчиками.

Получение сугубо технических решений из-за недостатка взаимодействия

Без сотрудничества ключевые идеи предметной области могут затеряться, а простые для понимания — выйти на первый план. Внутри организаций важные аспекты предметной области часто выражены нечетко; разработчики, пренебрегающие сотрудничеством со специалистами, могут упустить эти аспекты и сосредоточиться на «низко висящих плодах». В отсутствие взаимодействия, помогающего выявить новые идеи и удостовериться в правильном их понимании, разработчики будут вынуждены использовать отвлеченные технические термины и переводить их для бизнес-пользователей, чтобы они могли понять, как задачи в пространстве решений соотносятся с предметным пространством.

Основной целью сотрудничества является привлечение большого числа людей с разными точками зрения к работе над созданием модели предметной области, которую можно было бы использовать для решения поставленных задач. Никто не должен иметь исключительного права на истину, и никакое предложение нельзя считать глупым.

Попытка переработки знаний с кем-то, не являющимся специалистом в предметной области, может оказаться напрасной тратой сил. Бизнес-аналитик, действующий

щий как посредник между вами и специалистом, сможет передать вам требования и сыграть роль своеобразного устройства ввода/вывода, но он не сможет помочь с формированием модели, отвечающей сценариям использования.

Большие затраты времени на то, что не представляется важным

Разработчики должны понять основную причину, объясняющую, почему они создают новое программное обеспечение вместо сборки решения из готовых компонентов. Понимание стратегии компании в принятии решения о собственной разработке вместо покупки готовых решений поможет разработчикам максимально сконцентрироваться на процессе. Без концентрации на том, что является важным для успеха проекта, группы с ограниченными ресурсами не смогут грамотно распределить эти ресурсы с учетом наиболее важных аспектов — смысловых ядер. Выделение недостаточного количества ресурсов на наиболее важные направления в проекте является антишаблоном.

Архитектура программного обеспечения получится ущербной без ясного и общего понимания целей продукта, зачастую изложенных как введение в предметную область (domain vision statement). Правильные и обоснованные архитектурные решения возможны только в том случае, когда разработчики понимают предпосылки, стоящие за требованиями бизнес-пользователей. Непонимание этих предпосылок и разработка вслепую ведут к неудовлетворительным результатам.

Готовность признать, что не все системы *требуют* тщательного проектирования и не все системы *должны* детально прорабатываться, является большим прогрессом для группы разработчиков. Без концентрации на ключевых аспектах системы талантливые члены группы могут оказаться отвлечены на решение инфраструктурных задач и быть против реализации уровня представления на JavaScript вместо ключевых аспектов продукта.

Для специалистов в предметной области, как и для разработчиков, также важно ясно видеть смысловое ядро. Специалист, не разделяющий мнение заинтересованных лиц или не до конца понимающий, почему было принято решение написать новое программное обеспечение, не сможет оказать эффективную помощь в процессе переработки знаний из-за отрицательного отношения или недопонимания.

Превращение простых задач в сложные

Применение приемов, предназначенных для решения сложных задач, там, где сложность невысока, в лучшем случае приведет к напрасной трате сил, а в худшем — к созданию сложного решения, трудного в сопровождении из-за наложения абстракций. Философию DDD лучше использовать для создания стратегически важных приложений; в противном случае глубокие знания, приобретенные в результате применения DDD, не будут представлять большой ценности для организации.

Создавая систему, разработчики должны стремиться к простоте и ясности. Наполнение программного обеспечения абстракциями может лишь потешить эго разработчиков и замаскировать в действительности простой код. Разработчики, которые не увлечены созданием настоящей ценности для предприятия и сосредоточены только на технических амбициях, будут придумывать ненужные сложности из-за возрастающего нежелания решать предметные задачи. Такая конструкция программного обеспечения может вызвать отчаяние у тех, кто в будущем будет вынужден сопровождать «месиво» из технических наслоений.

Применение принципов DDD к простым областям, не имеющим большого значения для предприятия

Простые задачи требуют простых решений. Тривиальные области или подобласти, не имеющие стратегического значения для предприятия, не выиграют от применения всех принципов DDD. Разработчики, неукоснительно применяющие принципы DDD к любым проектам, независимо от сложности предметной области, наверняка встретят непонимание со стороны специалистов, не заинтересованных в детальной проработке областей, менее важных для предприятия.

DDD — это комплекс принципов, помогающих справиться со сложными предметными областями, имеющими большое значение для предприятия. Задачи, не связанные с большими ожиданиями со стороны предприятия, должны решаться в необременительной манере. Однако это не значит, что к их реализации можно подходить бессистемно. Напротив, код должен быть надежным, эффективным и простым в сопровождении, тогда как задачи, не имеющие сложной логики, должны иметь простые решения.

Большие и сложные системы будут иметь множество подразделов. Некоторые будут содержать сложную логику и иметь ключевое значение, другие будут просто управлять данными с применением простой логики. Тактические шаблоны DDD, наряду с формированием единого языка для обсуждения моделей, должны быть зарезервированы только для работы над основными областями. К ним относятся области, где ясность необходима, чтобы иметь возможность быстро вносить изменения, а также модели со сложной внутренней логикой. Разработчики не должны тратить на неспециализированные области или подобласти больше энергии, чем это необходимо для обеспечения их работоспособности, и изолировать их от смысловых ядер.

Отказ от шаблона CRUD как от антишаблона

Не все системы требуют тщательной проработки; попытка привести весь код в идеальное состояние может оказаться напрасной тратой сил. Все ваше внимание и силы должны быть направлены на смысловое ядро, для всего остального достаточно уровня «хорошо». Для систем, которые просто формируют наборы данных с использованием простой предметной логики или без нее, порой достаточно реализовать архитектуру CRUD с операциями создания, чтения, изменения и удаления, а освободившееся время потратить на реализацию смыслового ядра.

Использование шаблона «Предметная модель» для всех ограниченных контекстов

Шаблон «Предметная модель» хорош для реализации сложных или часто изменяющихся моделей. Усилия, потраченные на воплощение этого шаблона для неспециализированных моделей или моделей, не имеющих сложной предметной логики, стоят намного дороже, чем любые выгоды, которые эти усилия принесут. Используйте приемы проектирования на основе модели и шаблон «Предметная модель» только для смыслового ядра, а для более простых частей системы — другие шаблоны.

Спросите себя: нужна ли эта дополнительная сложность?

Когда начинающие разработчики осваивают новые шаблоны проектирования, они стремятся применять их к любому коду, который пишут. Такое поведение часто можно наблюдать в группах, недавно познакомившихся с DDD. Они концентрируются только на тактических шаблонах DDD, слепо используя их в любых проектах, независимо от того, оправданно ли это. Такое необоснованное стремление применить новую философию к любому программному обеспечению может привести к ненужным сложностям там, где достаточно простого решения.

Не следует писать программы, не имеющие стратегического значения, если достаточно возможностей стандартных программных продуктов. Если автоматизация ручного процесса требует слишком много усилий, просто оставьте этот процесс ручным. Помните: решения не всегда должны быть техническими.

Недооценка стоимости применения DDD

Следование принципам DDD является сложным и затратным по времени и ресурсам процессом. Поэтому применять данные принципы в полном объеме следует только к самым важным областям системы: к смысловому ядру. Принципы ложатся бременем на предприятие, вынуждая его работать вместе с вами на получение решения, а не на вас. Философия DDD часто оказывается наиболее ценной в областях, не связанных с технической реализацией.

Попытка добиться успеха без мотивированных и целеустремленных разработчиков

Философия DDD подходит не для всех. В том числе она подходит не для каждого проекта. Для получения максимальной выгоды от следования философии DDD вам потребуется сложное смысловое ядро, предполагающее длительное и циклическое развитие, а также доступ к специалистам в предметной области. Но это далеко не все, что может быть нужно. Для успешного и эффективного применения DDD необходим еще целый ряд условий, в частности:

- неукоснительное следование принципам проектирования, необходимое для рефакторинга;

- тонкое понимание проекта;
- целеустремленные, мотивированные и опытные разработчики.

Вам нужны дисциплинированные разработчики, которые готовы работать со специалистами в предметной области и вникать в работу предприятия, а не просто стремящиеся втиснуть в проект какой-нибудь новомодный фреймворк на JavaScript.

Помните: DDD не панацея от всех бед. Простой переход от каскадной модели разработки к методам гибкого проектирования или экстремального программирования не устранил все ваши проблемы, и решение следовать принципам DDD не поможет вам тут же начать производить более качественное программное обеспечение. Общим знаменателем любого успешного проекта является группа умных людей, влюбленных в свою работу и стремящихся к успеху.

Сотрудничество с незаинтересованными специалистами

Владение предприятием или заинтересованность в его развитии со стороны специалистов в предметной области является ключом к успешному сотрудничеству. Если разработчики будут работать со специалистами, не заинтересованными в успехе проекта, не понимающими его целей или не разделяющими его взглядов, они едва ли смогут помочь в создании эффективной модели и определении общего языка и едва ли смогут функционировать как единая слаженная команда. Разработчики играют ведущую роль в проектировании систем для решения задач предметной области. Однако в сотрудничестве со специалистами они смогут продвинуться гораздо дальше и вместе избавиться от необходимости создания программных решений путем устранения источников проблем и переосмысления бизнес-процессов.

Методологии итерационной разработки

DDD — это философия коллективного обсуждения идей и совместного обучения. Она призывает не останавливаться на первой рабочей идее, а продолжать экспериментировать, чтобы вскрыть, вероятно, более удачное решение или просто получить подтверждение того, что первая идея была более удачной. Все это требует времени, поэтому методологии, не разделяющие данную позицию, не поддерживают DDD.

Нельзя создать эффективную модель с первой попытки, поэтому для улучшения архитектуры проекта необходима итерационная методология разработки. Модели развиваются непрерывно. Разработчики, не понимающие, что модель и язык действительны только в данный отрезок времени, быстро заметят, что их творение превратилось в «большой ком грязи». Модель нужно любить. Ее нужно чистить и реорганизовывать по мере накопления новых знаний и появления новых сценариев использования.

Также следует отметить, что для безопасности экспериментов с моделью и ее дальнейшего развития необходим полноценный набор модульных тестов. Это не влечет необходимости применения методологии разработки через тестирование, но у вас должны быть гарантии того, что код сохранит работоспособность после его рефакторинга.

Применение DDD к любым задачам

Каждое решение должно соответствовать задаче. DDD — это философия проектирования, предназначенная для определенного предметного пространства. Этот инструмент стоит того, чтобы включить его в свой арсенал. Однако если предметная область не отличается сложностью или не подвержена частым изменениям, она автоматически становится непригодной для «молотка DDD»: не забывайте о существовании в вашем арсенале других, более подходящих инструментов. Используйте приемы моделирования и DDD только в самых сложных ограниченных контекстах, имеющих наибольшую ценность для вашего заказчика.

Не все подобласти одинаково сложны. Некоторые области или контексты могут вообще не требовать создания полноценной предметной модели и содержать лишь данные, для работы с которыми достаточно простых CRUD-операций. Для реализации простых контекстов используйте шаблоны CRUD/«Активная запись»/«Сценарий транзакции», а тактические шаблоны DDD оставьте для разделов системы, особенно важных для заказчика, имеющих сложную логику и часто изменяющихся.

Спросите себя, помогут ли дополнительные усилия получить решение или это только замедлит вашу работу. Стремитесь к простоте, но не упрощайте до банальности. Не усложняйте решение и не старайтесь задействовать бесполезные фреймворки. Достижение простоты — это искусство, требующее практики и прагматичного мышления.

Жертвование простотой ради ненужной чистоты

Не следует стремиться к идеалу в областях, где это не требуется. Сохраняйте неспециализированные области и области поддержки простыми и ясными. Используйте CRUD и простые шаблоны реализации несложной предметной логики. Пишите минимально работоспособный программный код, а затем переключайтесь на смысловое ядро. Именно смысловое ядро является той областью, где действительно нужно стремиться к идеалу. В ограниченных контекстах, не имеющих большой важности, иногда лучше оставить «маленькие комочки грязи» — это позволит быстро написать программный код и забыть про него. Если в дальнейшем понадобится внести какие-то изменения, вы сможете просто переписать этот фрагмент. Области, не важные для предприятия, едва ли будут изменяться или потребуют длительного развития, поэтому в таких областях работающий код предпочтительнее идеального. Часто лучшее — враг хорошего. Не беспокойтесь особенно о таком коде, не старайтесь найти подтверждение его правильности — это пустая трата сил и времени. Оставьте волнения о чистоте кода для областей, которые действительно заслуживают этого.

Пустая трата сил на поиски подтверждений правоты

Построив систему, которая следует всем принципам DDD, вы едва ли удостоитесь поощрительной грамоты от Эрика Эванса. Слепое следование любому языку шаблонов и любой методологии без учета уникальности собственного контекста —

чистой воды безрассудство. Стремление придерживаться ряда правил только ради самой методологии есть антишаблон. Философия DDD не призывает следовать определенным правилам или применять шаблоны программирования. Это процесс обучения. Сам процесс важнее цели, и он заключается в исследовании предметной области в сотрудничестве со специалистами, а не в подсчете числа шаблонов проектирования, которые вам удалось применить в решении.

Читайте форумы и статьи в блогах, где другие делятся опытом использования DDD и сотрудничества с бизнесом для обучения и совершения открытий. Не старайтесь создать идеальный шаблон репозитория и не ищите подтверждения правоты от коллег, не вовлеченных в работу над проектом и не имеющих полного представления о контексте, равно как не воспринимайте их советы всерьез.

Постоянное стремление к совершенству кода

Разработчики, одержимые стремлением применять шаблоны и принципы проектирования там, где это надо и где не надо, почти наверняка будут создавать сложные и запутанные архитектуры, упуская из виду главную цель. Разработчики должны понимать цели шаблонов проектирования и использовать их разумно. Слепое применение тактических шаблонов DDD не прибавит ценности для предприятия.

Гибкость архитектуры в важных областях, подверженных частым изменениям, помогает обеспечить гибкость модели и ее развитие без серьезных побочных эффектов. Активное стремление к совершенству архитектуры в областях, не имеющих большой ценности для предприятия и не предполагающих длительного развития, является пустой тратой сил. Лучше оставить «маленькие комочки грязи», изолированные от других контекстов, которые впоследствии легче заменить, чем стараться писать совершенный код повсюду.

При работе над смысловым ядром разработчики определенно должны выждать некоторое время, прежде чем утвердиться в выборе шаблона и образа мышления. Отложите реорганизацию кода и поэкспериментируйте с ним, чтобы увидеть, что вызывает трения/изменения, — это поможет глубже проникнуть в предметную область и сделать более обоснованный выбор.

Не отвлекайтесь на шаблоны, фреймворки или методологии; все это лишь детали реализации. Ваша цель — понять предметную область до самых глубин, чтобы лучше подготовиться к решению задач в ней. В этом заключается истинная ценность DDD.

Цель DDD — обеспечить ценность

Не позволяйте шаблонам и принципам проектирования мешать вам делать свою работу и создавать то, что представляет ценность для предприятия. Шаблоны и принципы должны помогать в создании гибких архитектур. Вы не получите орденов и медалей за их применение в приложении. Цель DDD — обеспечить ценность, а не произвести элегантный код.

Ключевые идеи

- Тактические шаблоны DDD могут привести к созданию эффективной предметной модели; однако это все еще развивающаяся область DDD, и детали реализации переоценены. Шаблоны могут иметь значение, но не они являются ценностью DDD.
- DDD — это больше чем философия программирования. Сотрудничество со специалистами в предметной области над переработкой знаний и создание единого языка, выражающего общее понимание предметной области, — вот основная идея DDD.
- Контексты решают все; контексты и изоляция обеспечивают целостность программного кода. Они уменьшают когнитивную нагрузку и делают модель более специализированной.
- Вам нужна команда умных и целеустремленных разработчиков, готовых изучать предметную область.
- Вам нужны специалисты в предметной области. Без их помощи невозможно глубокое проникновение в предметную область.
- Используйте шаблон CRUD для простых ограниченных контекстов. Если вы не определили предметную модель, это еще не значит, что вы плохой программист.
- Ограниченные контексты и единый язык являются основой DDD.
- Суть DDD заключается в процессе изучения, переработки, экспериментирования и исследований, чтобы в сотрудничестве вскрыть и определить эффективную модель.

10

Применение принципов, приемов и шаблонов DDD

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Как внедрить предметно-ориентированное проектирование
- Применение приемов и принципов предметно-ориентированного проектирования в проекте
- Осознание важности исследований и экспериментов на пути к эффективной модели
- Почему устранение неоднозначностей существенно увеличивает эффективность моделирования
- Устранение технологических сложностей при решении задач
- Почему не следует волноваться о создании идеальной предметной модели
- Как понять, что вы действуете правильно

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.worx.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 10 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

В предыдущих девяти главах вы получили общее представление о философии DDD. Эта глава объединяет все предыдущие знания, чтобы показать, как начать применять приемы и принципы DDD в проектах. Остальная часть книги будет сосредоточена на шаблонах программирования, помогающих реализовать эффективную предметную модель в коде, интеграции ограниченных контекстов и создании приложений, простых в сопровождении.

Внедрение DDD

Методология DDD не является панацеей от всех бед, и ее не следует воспринимать как таковую. Она не решит всех ваших проблем, равно как и методики гибкой раз-

работки; тем не менее это мощная и очень эффективная методология, если применять ее в соответствующих обстоятельствах, перечисленных ниже.

- У вас есть команда опытных, мотивированных и целеустремленных разработчиков, готовых учиться.
- У вас есть нетривиальная предметная область, имеющая большое значение для предприятия.
- У вас есть возможность общения со специалистами в предметной области, разделяющими ваши взгляды на проект.
- Вы следуете методологии итерационной разработки.

Без этих ключевых ингредиентов применение приемов и принципов DDD только усложнит процесс разработки вместо его упрощения. Однако если ваша ситуация подпадает под описанные выше условия, тогда применение приемов и принципов DDD может существенно увеличить ценность результата ваших усилий.

Не нужно внедрять DDD как методологию разработки проекта; вместо этого изучите принципы философии и применяйте их соответственно и только там, где они будут иметь значение. Подобно шаблонам проектирования, которые предпочтительнее применять в процессе реорганизации кода, приемы и принципы DDD должны использоваться только при необходимости, с учетом конкретных обстоятельств. Любые приемы должны применяться обоснованно и только в тех случаях, когда преимущества от их использования неоспоримы.

Обучение разработчиков

Обсуждая философию DDD в своей группе, сосредоточьте внимание не на языке шаблонов, а на ее соответствии потребностям предприятия, важности стратегических контекстов и языке описания модели. Стратегические шаблоны и принципы DDD намного важнее и полезнее, чем тактические шаблоны. Разработчики, сосредоточенные исключительно на тактических шаблонах программирования, упустят истинную ценность DDD и с большой долей вероятности потратят силы и время на применение приемов, не получив никакой выгоды.

Технические приемы не решают предметных задач, они лишь определяют детали реализации. Решение может дать только сотрудничество со специалистами в предметной области, владеющими ключом к раскрытию эффективной модели. Модели рождаются в коллективных обсуждениях, на карточках или на доске, и только потом реализуются в Visual Studio (или другой среде разработки). Помните: технические приемы могут усложнить решение задачи, концентрация же на абстрактной концептуальной модели поможет освободиться от инфраструктурных нагромождений и технических проблем, что позволит группе создать решение, отражающее суть предметной области и понятное специалистам предприятия.

Общение со специалистами

Заинтересованным лицам на предприятии не нужны слова о новой философии или методологии разработки, им нужно, чтобы вы создали для них нечто ценное. Все эти методики гибкой разработки, сервис-ориентированные архитектуры

(Service-Oriented Architecture, SOA) и облачные технологии слишком переоценены, они обещают решить все проблемы разработки, но в действительности на это не способны. Поэтому говорите только о своем желании узнать больше о работе предприятия, чтобы принести ему как можно больше пользы. Говорите о потребности ознакомить разработчиков с целями и задачами предприятия. Заинтересованные лица поймут важность общения разработчиков со специалистами предприятия и соответствие этого общения ожиданиям предприятия.

Философия DDD не работает без участия специалистов в предметной области. Если вам удастся убедить заинтересованных лиц в важности участия в разработке специалистов от предприятия, вы сделаете большой шаг к успеху. Конечно, в разговорах с заинтересованными лицами можно упомянуть философию DDD, но будет лучше, если вы сделаете упор на необходимость сотрудничества. Успех продукта во многом зависит от степени участия предприятия и его специалистов; только так можно внедрить DDD. Обучение разработчиков с целью более глубокого понимания логики предметной области позволит им создать более качественный продукт.

Применение принципов DDD

Только в сотрудничестве с заинтересованными лицами, понимающими ценность времени специалистов, и разработчиками, сосредоточенными на поиске решений сложных задач, окружающих модель предметной области, создаваемую совместно со специалистами в этой области, вы будете иметь все возможности для применения принципов DDD. В этом разделе описываются этапы подготовки проекта по разработке решения, а также то, какие приемы используются на каждом этапе. Однако ключ к применению DDD в том, чтобы начать с самого простого. Начинайте с решения самых простых задач, пока не столкнетесь со сложностями или неоднозначностями. Столкнувшись с неоднозначностью во время общения, явно определите ее в едином языке (UL). Когда модель станет слишком большой, разделите ее на части и примените стратегические шаблоны организации кода и приемы моделирования. Если вы будете стремиться к простоте и добиваться применения приемов и принципов, вместо того чтобы колоть орехи кувалдой, вы получите огромные выгоды от DDD.

Понимание замысла

Перед определением требований важно ознакомить группу разработчиков с ожиданиями заинтересованных лиц. Начните встречу с заинтересованными лицами с выяснения причин, по которым было принято решение создать новое программное обеспечение вместо покупки готовых программных продуктов, а также с представлений о будущем продукте. В этом вам помогут следующие вопросы:

- Что стало побудительным мотивом к созданию данного продукта?
- Какое значение для предприятия имеет этот продукт?
- Как определить успешность продукта? По каким признакам?
- Чем он должен отличаться от того, что было прежде?

Выслушав ответы заинтересованных лиц на данные вопросы, вы сможете определить наиболее важные аспекты продукта — область применения программного обеспечения, являющуюся важнейшей причиной его создания. Запишите эту информацию и изложите на бумаге предметную концепцию, чтобы обозначить общую цель для вашей команды. Члены команды должны понимать, что будет способствовать или препятствовать успеху продукта, что является особенно важным и в чем заключается главная ценность.

В ходе встречи может выясниться, что продукт не является чем-то особенным и существует стандартное решение, покупка которого обойдется дешевле, чем создание собственного. В такой ситуации четко укажите на это обстоятельство и выясните, важно ли это для предприятия и его будущего. Может случиться и так, что у вас самих уже будет готов прототип продукта для оценки заинтересованности заказчиков. Все это поможет вам понять, в чем заключается ценность и какие цели преследует предприятие. С этим пониманием вы сможете определить, какие приемы DDD окажутся эффективными для вашего проекта.

Определив назначение продукта и цели заинтересованных лиц, вы сможете сформулировать характеристики продукта, соответствующие общему замыслу.

Определение требуемой функциональности

Эффективный способ вовлечения заинтересованных лиц в процесс определения требований заключается в применении методологии разработки через реализацию поведения (BDD). BDD — это общий язык, помогающий определить поведение системы. Его можно считать аналогом единого языка для требований. Он позволит разработчикам понять требования, предъявляемые заинтересованными лицами, которые не являются техническими специалистами. Применение BDD и определение требований на языке, понятном работникам предприятия и явно выражающем ценность продукта, может служить отличным упражнением, демонстрирующим возможность наладить общение с помощью общего языка. Этот процесс поможет показать команде силу языка, а также то, к чему могут привести неоднозначности. Используйте эти встречи для прояснения терминологии. Такое упражнение подготовит членов команды к будущей работе над созданием единого языка для описания модели, реализующей правила, логику и процессы, необходимые для удовлетворения требований к поведению системы.

От заинтересованных лиц вы узнаете бизнес-сценарии использования, входные данные системы и то, какой результат она должна производить. На основе бизнес-сценариев использования будут формироваться ваши прикладные службы. От их сложности будет зависеть выбор шаблона для реализации предметной логики. В ходе сбора требований сосредоточьтесь на пожеланиях заинтересованных лиц и причинах этих пожеланий. Причины в данном случае особенно важны. Дополнительные вопросы помогут выяснить, чем именно обусловлены такие пожелания заинтересованных лиц. Выясняя требования, оставайтесь в предметном пространстве и сосредоточьтесь вниманием на возможностях, которыми должен обладать данный продукт. Вопрос, как удовлетворить требования, может подождать, пока вы не поймете замысел и не придете к взаимопониманию.

Я встречался с командами, галопом проскакивавшими этап сбора требований и устремленными к скорейшей реализации решения без детального исследования задачи. Не торопитесь приступать к решению, внимательно исследуйте предметное пространство вместе с заинтересованными лицами. Часто они не могут ясно выразить свои пожелания. Исследуя предметное пространство, вы сможете вытянуть из них истинные потребности и даже предложить лучшие варианты поведения системы, не тратя времени на реализацию решений, которые на самом деле не нужны предприятию.

В процессе заполнения карточек пользовательских историй и определения предполагаемых функций может обнаружиться, что продукт получается слишком громоздким для управления или же в предметном пространстве присутствуют неоднозначности. Вы можете также потерять видение общей картины или основных причин создания данного продукта. В таких случаях необходимо провести дистилляцию предметного пространства.

Дистилляция предметного пространства

Если предметное пространство стало слишком обширным, можно попробовать уменьшить когнитивную нагрузку, абстрагировав задачу на более высоком уровне путем создания подобластей. Часто полезно рассмотреть, поддержку каких подразделений предприятия должен обеспечить продукт, и на этой основе выделить подобласти. Подразделения предприятия осуществляют деятельность, необходимую для всего бизнес-процесса; рассмотрите эти виды деятельности без привязки к организационной структуре или функциям. Среди них будут присутствовать неспециализированные или поддерживающие виды деятельности. Те же, что представляют истинный интерес и способны обеспечить успех продукта, являются смысловым ядром.

Концентрация на важном

Раздробив предметное пространство на подобласти, уделите больше времени обсуждению с заинтересованными лицами поведения смыслового ядра. Смысловое ядро может оказаться очень маленьким, и это замечательно. Обсуждение этой области требует особого внимания, потому что она представляет наибольший интерес и имеет наибольшую ценность для вас. Смысловое ядро должно прямо соответствовать общим представлениям заинтересованных лиц; если это не так, возможно, вы неправильно идентифицировали смысловое ядро или вам стоит уточнить представления заинтересованных лиц.

Понимание действительной картины

После изучения предметного пространства и выяснения, в чем заключается истинная ценность системы, можно приступать к моделированию решения. Однако прежде чем начать создавать решение для любого проекта, чрезвычайно важно понять, в каком окружении вам предстоит работать. Знание, какие программные решения уже используются на предприятии, поможет принять обоснованные

решения по интеграции вашего продукта. Самый лучший способ охватить общую картину — создать карту контекстов.

Разработчики должны определить ограниченные контексты, непосредственно затрагивающие ваш продукт или затрагиваемые им. Разработчики также должны определить, как взаимодействуют эти контексты, какие протоколы используют и какими данными управляют. Для этого рекомендуется выполнить следующие шаги.

1. Определите, какие модели, непосредственно затрагивающие предметную область или затрагиваемые ею, известны команде. Нарисуйте эти контексты, дайте им названия и отметьте, кто за них отвечает. Если вы не знаете, с чего начать, взгляните на организационную структуру области, в которой работаете, потому что большинство систем строится вокруг взаимодействий между подразделениями. Затем взгляните на организационную структуру подразделений, занимающихся разработкой.
2. Отметьте на карте точки и методы интеграции.
3. Укажите, какие данные участвуют во взаимодействиях и кому они принадлежат.
4. Подпишите отношения между контекстами. По отношению к каким группам ваша группа является «нижней» и на какие их решения она должна опираться? Для каких групп ваша является «верхней» и кому она должна сообщать о решениях и изменениях?
5. Смыывайте и повторяйте¹, пока не зафиксируете все, что нужно знать об окружении, в котором предстоит работать.

Все разработчики в группе должны уметь читать и понимать карту контекстов. Повесьте ее на стене, чтобы каждый мог видеть ее. Это ваша карта военных действий. Она должна быть достаточно простой, чтобы любой мог быстро разобраться в ней, поэтому не тратьте слишком много времени на создание диаграмм UML. Вместо этого нарисуйте общую панораму. Детализировать нужные точки и вникать в детали вы начнете, когда перейдете к вопросам интеграции.

Моделирование решения

Прежде чем начать моделировать решение и применять принципы DDD, нужно убедиться, что продукт, для которого вы собираетесь создать решение, отвечает следующим критериям.

- Является сложной задачей или имеет сложные подобласти.
- Важен для предприятия и на него возлагаются большие ожидания.
- Имеется доступ к специалистам в предметной области.
- Подобрана команда умных и мотивированных разработчиков.

¹ Намек на анекдот про программиста, который надолго застрял в душе, потому что следовал инструкции на шампуне: «Намыльте, смойте и повторите». — *Примеч. пер.*

Если задача, стоящая перед вами, или ее часть не отличается сложностью, тогда создание предметной модели может оказаться излишеством. Когда объем предметной логики невелик и вся суть заключается в простом управлении данными, следует выбрать менее сложный шаблон реализации предметной логики, такой как «Сценарий транзакции» или «Активная запись». При таком подходе вы сможете избежать использования более дорогостоящих методик.

Если продукт не слишком важен для предприятия и на него не возлагаются большие ожидания, возможно, не стоит прикладывать больших усилий, чтобы создать решение, способное выдержать проверку временем. Создайте достаточно хорошее решение, которое будет легко заменяться в будущем и не потребует серьезных вложений. Можно попробовать использовать стандартное решение. Если задача является достаточно общей, вполне возможно, что найдется открытое решение, отвечающее вашим потребностям.

Если у вас нет выхода на специалистов в предметной области, открытия в предметной области станут невозможными. Разработчики будут вынуждены строить абстракции на основе технических проблем, а язык программного кода не будет отражать предметные понятия. Контексты будут иметь размытые границы, и код быстро начнет превращаться в «ком грязи».

Если разработчики плохо мотивированы или испытывают недостаток знаний шаблонов и принципов проектирования корпоративных приложений, возможно, будет лучше отдать предпочтение более простым шаблонам организации предметной логики, чтобы чрезмерно не усложнять разработку.

Все задачи имеют разную сложность

Не все разделы предметной области требуют применения всего спектра приемов и принципов DDD. Вы должны выбрать основные направления. Если для реализации решения не требуются консультации специалистов в предметной области, не привлекайте их. Если для реализации решения в какой-то области не требуется использовать шаблон «Предметная модель» для представления абстрактной модели в коде, не используйте его.

Если какая-то часть предметного пространства отличается высокой сложностью, на нее возлагаются большие ожидания со стороны предприятия, имеется возможность привлечь специалистов в предметной области и разработчики готовы принять вызов, значит, ваш случай в полной мере соответствует критериям, согласно которым разработка продукта по методологии DDD окажется наиболее эффективной.

Привлекайте специалистов

Специалист в предметной области — это эксперт, обладающий глубокими знаниями в данной области. Если заинтересованные лица определяют, что должна делать система, то специалист в предметной области помогает разработчикам своими знаниями и опытом, чтобы совместно прийти к модели решения, удовлетворяющего требованиям. Специалистом в предметной области может быть опытный

пользователь текущей системы, глубоко знающий процессы и логику предметного пространства. Точно так же специалистом в предметной области может быть разработчик текущей системы или просто тот, кто давно работает в отделе и занимался предметной областью много лет. Специалист — это не человек, обладающий званиями и регалиями; специалистом может быть любой работник предприятия, разбирающийся в предметной области.

Философия DDD не работает без специалистов. Это просто, как дважды два. Без специалистов не удастся проникнуть в суть предметной области, получить все богатство знаний и создать единый язык. В отсутствие специалистов разработчики могут обращаться за советами к пользователям существующей системы, но, несмотря на знание ими текущих процессов, они не смогут рассказать обо всех правилах игры или особенностях предметной области. Невозможно переоценить важность выбора специалиста и его вовлечения в работу. Специалист должен выполнять свою работу, его время дорого, поэтому с умом используйте время, которое он выделит на общение с вами.

Мнения бизнес-аналитиков также имеют ценность. Они обладают опытом, которым, возможно, не обладают разработчики и специалисты. Бизнес-аналитики могут организовать общение со специалистами и помочь разработчикам усвоить терминологию, необходимую для формирования единого языка.

Важно, чтобы заинтересованные лица доверяли специалистам и расценивали их как экспертов. Также важно, чтобы специалисты понимали причины разработки проекта и его цели. Специалист, не разделяющий видение сущности проекта, скорее будет мешать, чем помогать. Однако его суждения и доводы относительно проекта могут оказаться оправданными. В этом случае постарайтесь организовать обсуждение совместно с заинтересованными лицами и специалистом, чтобы избавиться от страхов и сомнений.

Постарайтесь разместить разработчиков поближе к специалистам предприятия. Разработчики должны иметь возможность легко и просто обращаться к специалистам и пользователям, чтобы обеспечить постоянную обратную связь. Специалисты в предметной области — ваш главный источник знаний, на основе которых вы сможете удостовериться в правильности предметной модели. Извлеките из голов специалистов столько информации, сколько сможете. Благодаря знаниям специалистов вы сможете создать более эффективную модель.

Реализуйте поведение и модель на основе конкретных сценариев

Работая в пространстве решений, сосредоточьтесь на удовлетворении требований к поведению продукта, а не на попытке смоделировать всю предметную область. Руководствуйтесь при создании модели выбранным поведением и определяйте конкретные бизнес-сценарии для использования в качестве примеров. Следуя этому правилу, разработчики и специалисты в предметной области смогут создать модель, соответствующую решаемой задаче. Такая практика не позволит не в меру ретивым разработчикам создать «одну-модель-на-все-случаи», которая в действительности не отвечает потребностям системы и является скорее отражением реальности, а не эффективной ее абстракцией.

Для примера рассмотрим реализацию поддержки купонов в области электронной коммерции:

Чтобы стимулировать спрос

Как владелец магазина

Я хочу предложить купоны на скидку

Прежде чем приступить к моделированию этой возможности, необходимо определить конкретные бизнес-сценарии ее использования. Для данной возможности можно определить несколько сценариев. Ниже приводится описание одного из них:

Сценарий: Покупатель получает скидку на заказ.

У него есть купон, дающий право на 10%-ную скидку от суммы заказа.

Действие купона распространяется на заказы, стоимость которых превышает \$50.

Когда купон применяется к заказу на сумму \$60, скидка должна составить \$6.

В процессе переработки знаний разработчики должны прислушиваться к языку, употребляемому специалистом в предметной области, и запоминать понятия, используемые при описании сценария. Если разработчики заметят потенциальные проблемы с моделью, они должны разрешить их с помощью специалиста.

Сотрудничайте со специалистами при работе над наиболее интересными разделами

Выбирая сценарии для моделирования, не старайтесь сорвать «низко висящие плоды»; оставьте в стороне простое управление данными. Вместо этого займитесь наиболее сложными разделами — интересными областями, расположенными в глубине предметной области. Сосредоточьтесь на разделах продукта, делающих его уникальным; они будут сложными и, возможно, на их изучение вам потребуется дополнительное время. Это время не будет потрачено даром — именно в этих областях сотрудничество со специалистами может оказаться наиболее эффективным. Обсуждение простых операций создания, чтения, изменения и удаления (Create, Read, Update, Delete, CRUD) данных быстро наскучит специалистам, они потеряют интерес и не захотят уделять вам свое время. Моделирование сложных областей, составляющих основу продукта, — вот для чего разрабатывались принципы DDD.

Создавайте единый язык для устранения неоднозначностей

Неоднозначности наряду с незнанием предметной области являются худшими врагами разработчиков. В процессе моделирования и переработки знаний разработчики должны четко определить все термины и сформировать единый язык, чтобы специалисты в предметной области могли пользоваться привычной терминологией. Все должны говорить на одном языке и иметь общее понимание идеи. Единый язык должен использоваться не только для общения со специалистами, — программный код также должен создаваться с применением тех же терминов и понятий, чтобы программные модели отражали модели ментальные, используемые при общении.

Единый язык формируется в процессе совместной переработки знаний специалистами и разработчиками, как только они приступают к моделированию решений для наиболее важных и интересных частей продукта. Ясность и однозначность в общении между разработчиками и специалистами имеют большое значение для открытий и уменьшают вероятность недопонимания из-за необходимости перевода понятий с языка программной модели разработчиков на язык ментальной модели специалистов. Если разработчики будут разговаривать со специалистами о шаблонах, приемах и принципах проектирования, последние быстро потеряют интерес из-за необходимости осваивать ненужную им терминологию и переводить с одного языка на другой. Модель, хотя и реализуется в программном коде, должна обсуждаться на более высоком уровне абстракции, чтобы специалист смог поделиться своими знаниями и опытом в решении проблем в каждом новом сценарии, предъявляемом ему.

По мере углубления знаний о предметной области будет развиваться и ваш единый язык. Вместе с языком должен развиваться и программный код. Проводите периодическую реорганизацию своего кода, чтобы учесть развитие языка и задействовать в коде более точные и говорящие имена методов. Если обнаружится, что начинает образовываться блок слишком сложной логики, расскажите специалисту, что делает этот код, возможно, он предложит более удачное предметное понятие. Если это получится, выделите такой код в отдельную спецификацию или класс правил.

ЧТО ТАКОЕ СПЕЦИФИКАЦИЯ?

Спецификация представляет бизнес-правила, которым полностью или частично должна соответствовать предметная модель. Спецификации можно также использовать для описания критериев запросов. Например, можно запросить все объекты, соответствующие заданной спецификации.

Отбросьте первую модель и создайте вторую

Начиная работу над новым проектом, вы почти ничего не знаете о нем, но именно в это время вы будете принимать важные решения. Ваша первая модель почти наверняка будет неправильной, но это не повод опустить руки. Новые знания о предметной области приобретаются в ходе множества итераций. Ваши познания будут углубляться, и на их основе вы сможете развить свою модель в нечто более полезное и ценное.

После создания первой модели многие группы разработчиков обычно прекращают исследования и усаживаются за клавиатуру, чтобы реализовать ее. Едва ли ваша первая модель сразу окажется лучшей. Создав неплохую модель, вы должны отложить ее и продолжить исследование задачи с другого направления. Исследования и эксперименты жизненно необходимы для более глубокого изучения предметной области, поэтому ошибайтесь и подтверждайте хорошие идеи, сравнивая их с плохими.

Иногда, занимаясь моделированием, вы можете зайти в тупик; ваше решение может оказаться не в состоянии удовлетворить новому сценарию с применением

текущей модели. Это замечательно. Вместо попытки подогнать сценарий под модель, создайте новую модель, которая успешно справится с прежними и новыми сценариями. Попробуйте забыть все, что вы узнали в процессе создания первой модели, и пойти другим путем. Исследуйте и экспериментируйте, чтобы приобрести новые знания и предложить новое решение.

Результатом рассмотрения задачи под разными углами является не создание идеальной модели, а вскрытие и изучение новых понятий предметной области. Это намного более ценно, так как поможет разработчикам создавать эффективные модели в каждой итерации.

Реализуйте модель в коде

После определения модели для сложной предметной области в сотрудничестве со специалистами ее правильность необходимо доказать, воплотив в программный код. Ментальная модель, созданная вами и специалистами, должна переноситься в код с применением той же терминологии, языка и понятий. После превращения ментальной модели в код разработчики могут увидеть, что она не удовлетворяет потребностям сценария и необходимо найти новую идею или изменить прежнюю. Благодаря наличию общего языка и общего понимания модели, общение со специалистами будет протекать легко и вы без лишних сложностей согласуете с ними новое решение. Изменения в коде должны быть отражены в ментальной модели, и обе модели должны развиваться вместе.

Создание предметной модели

Создание предметной модели — это циклический процесс, и поиск эффективной модели продолжается снова и снова с появлением новых задач, которые она должна решать. Важно не пытаться смоделировать всю предметную область, а отобрать тщательно продуманные сценарии использования, которые можно было бы использовать как пример для тестирования любой созданной модели.

Предметная модель должна постоянно соответствовать следующим двум принципам:

Уместность: способна отвечать на вопросы в предметной области и проводить в жизнь правила и требования.

Выразительность: способна сообщить разработчикам, *что* она делает и *зачем* она это делает.

С первой попытки нельзя создать законченную эффективную модель. На практике начальное воплощение предметной модели получается упрощенным, она не отражает богатства предметной области. Поэтому необходимо постоянно производить реорганизацию кода, чтобы отразить в нем новые знания предметной области.

Создание и развитие эффективной модели осуществляются в ходе исследований и экспериментов. BDD и разработка через тестирование (Test-Driven Development, TDD) помогают экспериментировать, давая уверенность, что соседние области не будут затронуты. Начните с реализации анемичной предметной модели или с еще

более простого шаблона и переходите к шаблону предметной модели, когда это покажется целесообразным. Создавайте модели, только когда задача требует комплексного решения или когда разработчики впервые занимаются данной предметной областью (например, никогда прежде не работали в финансовой сфере).

Сохраняйте решение простым, а код скучным

Сохраняйте модель максимально простой и конкретной, а программный код — скучным и легкочитаемым. Многие группы разработчиков быстро попадают в ловушку чрезмерного усложнения задачи. Сохранение решения простым означает не его небрежность, а отсутствие беспорядка и ненужной сложности. Способствуйте упрощению кода, проводя обзоры кода или применяя методику парного программирования. Разработчики должны соревноваться друг с другом в простоте решений и точном их соответствии только решаемой задаче, а не более широкому кругу похожих задач.

Ограничьте область для безопасности

Если вы занимаетесь доработкой унаследованного кода или интегрируете свое решение с унаследованным кодом, очень важно не допустить проникновения уже существующего беспорядка в ваш код. (Если, конечно, беспорядок действительно существует; не забывайте, что «унаследованный» не означает «плохой»!) Может показаться заманчивым провести чистку унаследованного кода, но эта задача быстро может стать слишком трудоемкой и отвлечет вас от фактической цели — создания новых функциональных особенностей. Вместо этого задействуйте шаблон «Предохранительный слой», чтобы создать границы между новым и существующим кодом. Такая защита границ позволит создать ясную модель, изолированную от влияния со стороны других групп разработчиков.

Интегрируйте модель раньше и чаще

В ходе моделирования очень важно как можно раньше убедиться в правильности модели. Диаграммы на доске и заполненные карточки — вещь, несомненно, хорошая, но истинным свидетельством движения вперед является только действующий код. Действующий код — это интегрированный код. Действующий код соединяется с базами данных и пользовательскими интерфейсами, он доказывает работоспособность модели от начала и до конца.

Нетехническая реорганизация

Опытные разработчики знают, как выполнять техническую реорганизацию (или технический рефакторинг) для повышения качества программного обеспечения путем перехода к хорошо зарекомендовавшим себя шаблонам организации кода. Однако необходим еще один вид реорганизации, чтобы модель ясно отражала, что именно она делает. Важно отразить в коде любые открытия в предметной области, которые случаются со специалистами и всеми, кто работает над программным продуктом. Код, реализующий предметную модель, должен быть ясным и выразительным.

Первоначально предметная модель основывается на поверхностных представлениях о предметной области, обычно на глаголах и существительных из описания требований или из начальных дискуссий со специалистами в предметной области. Однако с углублением знаний в каждой итерации должно расти и многообразие предметной модели. Даже если техническая архитектура кода изменится для увеличения его ясности, имена методов классов, а также пространств имен должны продолжать изменяться, чтобы соответствовать более точным абстракциям, обнаруженным в ходе переработки знаний. Такое непрерывное развитие модели поможет сохранить ее уместность, соответствие представлениям и целям специалистов в предметной области.

Разделяйте пространство решений

С ростом модели будут обнаруживаться неоднозначности в языке или перегруженные термины, а может, вы просто посчитаете, что такой моделью слишком сложно управлять из-за увеличившегося размера. Чтобы сделать большие и сложные предметные модели более простыми и легкими в сопровождении, выполните деление по контекстам, опираясь на естественный язык и требования. Особое внимание уделите минимизации связей между контекстами. Не стремитесь получить идеальный код, стремитесь получить идеальные границы. Ограниченный контекст и агрегаты — вот мощные понятия в DDD, которые помогут уменьшить сложность и справиться с ней.

Стратегические шаблоны DDD следует применять при превышении некоторого предела сложности, а не для ее упреждения. Удаление границ будет затруднено после их оформления, поэтому проводите их не ранее чем через несколько итераций разработки, когда вы накопите больше знаний о предметной области. Отдавайте предпочтение небольшим модулям кода, акцентируя внимание на границах, которые можно заменить, а не на создании идеального кода в этих границах. Изолируйте и защищайте данные.

Смывайте и повторяйте

На рис. 10.1 изображены шаги применения приемов и принципов DDD.

Модель постоянно развивается и изменяется; вы не сможете эффективно использовать приемы и шаблоны DDD, не учитывая это развитие. Вы должны снова и снова повторять шаги, представленные в этом разделе. Удержание сложности программного решения на как можно более низком уровне — вот главная цель DDD. Все приемы и принципы направлены на достижение этой цели. Ваша работа как разработчика состоит в том, чтобы постоянно проверять эффективность и простоту архитектуры модели со сменой итераций и ее развитием, в соответствии с новыми правилами поведения, установленными заинтересованными лицами.

Путь к эффективной модели лежит через сотни мелких реорганизаций кода. Изменения и уточнения вносятся в модель постоянно, по мере появления новых знаний.

В ходе разработки может обнаружиться, что предположения о смысловом ядре существенно изменились. Бизнес может прийти к выводу, что прежняя организация

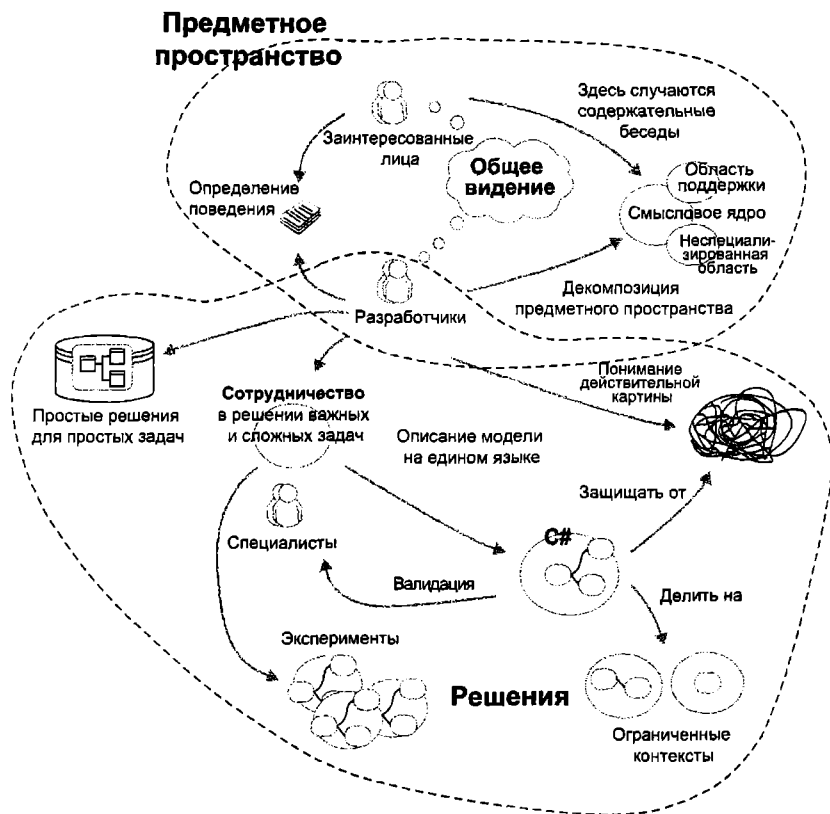


Рис. 10.1. Процесс DDD

работы была неправильной, и поменять многое. Ваши границы также изменятся, так как новая организация бизнеса сделает недействительной вашу архитектуру. С появлением новых особенностей могут возникнуть неоднозначности. В этом случае разбейте модель и явно определите две части в конкретных контекстах.

Начинайте с простого и используйте приемы и принципы, только когда они действительно необходимы. В противном случае есть риск чрезмерно усложнить простое решение простой задачи.

Исследования и эксперименты

Богатая возможностями и полезная модель является результатом исследований и творчества. Экспериментирование помогает взглянуть на код с разных сторон. Если обнаружится, что идеи с трудом реализуются в коде, значит, что-то было сделано неправильно. Не останавливайтесь на первой же работающей модели. Так как модель развивается в течение нескольких итераций, хорошо отложить реорганизацию до момента, пока не появятся новые знания о предметной области. Пусть

модель живет и развивается какое-то время. Вы не получите правильный результат с первого раза. Эксперименты и исследования — вот движущая сила изучения.

Ставьте свои предположения под сомнение

В каждой итерации разработчики должны ставить свои предположения под сомнение, потому что появление новых требований может означать, что прежде полезная модель больше соответствует решаемой задаче. Этот навык позволит создать гибкое программное обеспечение, развивать его с развитием продукта и предотвратить его превращение в «большой ком грязи» (Big Ball of Mud, BBoM).

Группа не должна слишком привязываться к модели, основанной на требованиях из первой итерации. В последующих итерациях может обнаружиться, что модель не соответствует вновь появившимся требованиям. Группы, не проявляющие гибкого отношения к развитию, очень быстро обнаружат, что код начал работать против них. Когда разработчики следуют методологии разработки через тестирование, они не останавливаются, достигнув работоспособной системы. Вместо этого они обеспечивают прохождение тестов и затем реорганизуют архитектуру, чтобы сделать ее более выразительной. Это, пожалуй, самый важный аспект разработки через тестирование и достаточно ценный для применения в DDD. Реорганизация с появлением новых знаний поможет произвести более выразительную и открытую модель.

Моделирование — это текущая работа

Моделирование должно выполняться по мере необходимости — это не определенный этап в методологии проектирования. Вы должны прерываться и вступать в диалог со специалистами, когда это потребуется. Если область понятна до мелочей, вы можете обнаружить, что модель вообще не нужна. Не привязывайтесь слишком к своему программному обеспечению, будьте готовы выбросить свой лучший код, когда изменится предметная модель. Каждая новая итерация несет новые вызовы. Вы должны быть уверены, что ваша модель готова к возможной реорганизации под новые потребности и сценарии использования.

Не существует неправильных моделей

Нет таких понятий, как глупый вопрос или глупая идея. Неправильные модели помогают подтвердить правильность полезных, а процесс их создания помогает изучению. При работе над сложным разделом или смысловым ядром предметной области разработчики должны быть готовы взглянуть на проблему под другим углом, рисковать и не бояться перевернуть ее с ног на голову.

Для сложного смыслового ядра разработчики должны произвести хотя бы три модели, чтобы получить возможность создать что-то полезное. Группы, не испытывающие частых неудач и не генерирующие множества идей, почти наверняка не проявляют должного усердия в работе. Даже в периоды активного общения со специалистами в предметной области разработчики не должны прекращать

обсуждение первых признаков чего-то полезного. Оказавшись в нужном месте, сотрите с доски все, что там было нарисовано, начните заново, взглянув на проблему под другим углом, и попытайтесь найти другой путь к решению. Находясь в одной комнате со специалистами, разработчики не должны покидать ее, пока не исчерпают все свои идеи.

УЧИТЕСЬ ЗАБЫВАТЬ

Не привязывайтесь к идеям; только методом проб и ошибок можно сформировать представления о предметной области, которые помогут решить задачи. Пишите тестовый код и тестируйте, пока не будете удовлетворены его архитектурой; вам будет проще конструировать решения и отбрасывать бесполезные модели, еще не включенные в состав приложения.

Податливый код способствует раскрытию

Во второй, третьей и четвертой частях книги вы познакомитесь с шаблонами организации программного кода, которые помогут упростить его изменение с появлением новых требований. Такой податливый код получается в результате многократных мелких реорганизаций. Постоянная реорганизация по мере углубления знаний помогает получить податливую архитектуру и гибкий код, облегчающие внесение изменений под новые требования к системе, появляющиеся в каждой новой итерации. Если модель жесткая и неподатливая, она бесполезна. Мартин Фаулер так отмечает важность принципов моделирования в своей книге «Analysis Patterns: Reusable Object Models»: «Проектируйте модель так, чтобы самые частые изменения модели затрагивали меньшее число типов».

Но опасайтесь преждевременного выполнения реорганизации. Не приступайте к ней, пока не накопите достаточно знаний о предметной области и не утвердитесь в мысли о необходимости применения шаблонов проектирования. Отсрочка реорганизации может также показать, какие области кода изменяются чаще и почему. С этим знанием вы более обоснованно подойдете к изменению архитектуры программного кода.

Превращение неявного в явное

Когда разработчики глубоко погружаются в код, они часто не замечают или пропускают логические инструкции как простые артефакты программирования. Эти маленькие неявные блоки кода скрывают важные детали предметной области, часто маскируя их важность. Если такие архитектурные решения не сделать явными, они не будут добавлены в ментальную модель и дальнейшие открытия будут даваться с большим трудом.

Как отмечалось выше, отсрочка реорганизации может способствовать вскрытию важных деталей в коде и, соответственно, важных деталей в модели. Если вдруг вы обнаружите блок кода, реализующий некоторую предметную логику, пока не

имеющую названия, сообщите об этом специалисту, дайте название логическому понятию и заверните программный код в это понятие. Очень важно своевременно превращать подразумеваемые понятия в явные. Любые решения, принимаемые в программном коде, должны явно сообщаться специалисту и оформляться в модели в виде явных понятий.

Борьба с неоднозначностями

Часто случается так, что специалисты не говорят о понятиях или только намекают на те из них, которые могут послужить ключом к более глубоким открытиям в предметной модели. Такие подразумеваемые понятия, выглядящие второстепенными для специалиста, должны быть обозначены явно, они должны получить названия и быть полностью понятны разработчикам. Например, рассмотрим сайт электронной коммерции, не позволяющий зарубежным покупателям включать в заказ более 50 пунктов. Разработчики легко могут реализовать это бизнес-правило, как показано в листинге 10.1.

Листинг 10.1. Неявная логика в коде

```
public class basket
{
    private BasketItems _items;

    // ...

    public void add(Product product)
    {
        if (basket_contains_an_item_for(product))
        {
            var item_quantity = get_item_for(product).quantity()
                               .add(new Quantity(1));

            if (item_quantity.contains_more_than(new Quantity(50)))
                throw new ApplicationException(
                    "You can only purchase 50 of a single product.");
            else
                get_item_for(product).increase_item_quantity_by(
                    new Quantity(1));
        }
        else
            _items.Add(BasketItemFactory.create_item_for(product, this));
    }
}
```

Однако спустя месяцы другие разработчики могут не понять причину такого правила. Поэтому вам следует понять, что стоит за этим правилом, и дать фрагменту кода соответствующее имя. Как известно, поставщики выдвигают такое требование, чтобы исключить возможность оптовой торговли через подобные сайты. Выяснив причину, можно явно сослаться в программном коде на это правило,

завернув его в класс, указывающий на глубокое понимание предметной области. Реорганизованный код представлен в листинге 10.2.

Листинг 10.2. Явная логика в коде

```
public class basket
{
    private BasketItems _items;

    // ...

    public void add(Product product)
    {
        if (basket_contains_an_item_for(product))
        {
            var item_quantity = get_item_for(product).quantity()
                               .add(new Quantity(1));

            if (_over_seas_selling_policy.is_satisfied_by(item_quantity))
                get_item_for(product).increase_item_quantity_by(
                    new Quantity(1));
            else
                throw new OverSeasSellingPolicyException(
                    string.format(
                        "You can only purchase {0} of a single product.",
                        OverSeasSellingPolicy.quantity_threshold));
        }
        else
            _items.Add(BasketItemFactory.create_item_for(product, this));
    }
}

public class OverSeasSellingPolicy
{
    public static Quantity quantity_threshold = new Quantity(50);
    public bool is_satisfied_by(Quantity item_quantity, Country country)
    {
        if (item_quantity.contains_more_than(quantity_threshold))
            return false;
        else
            return true;
    }
}
```

Разработчики должны следить за появлением неоднозначностей или несогласованностей в коде или в языке, которым пользуются специалисты. Особое внимание также следует уделять языку, которым пользуются другие члены команды при обсуждении предметной модели. Всегда подтверждайте свои предположения, касающиеся языка и особенностей модели, общаясь со специалистами. Подтверждайте ясно, четко, во всеуслышание, и проектируйте лингвистически согласованные решения. Если специалист не использует какой-то термин, этот термин не должен присутствовать ни в языке, ни в программном коде. Если термин в модели

потерял свое значение или стал бесполезен, удалите его. Язык должен оставаться небольшим и специализированным. Специалисты должны возражать против неподходящих терминов или структур в языке или в модели.

Давайте названия

Если специалист говорит о чем-то, превратите это «что-то» в явное понятие. Если специалист намекает на что-то, сделайте это «что-то» явным. Если специалист не может понять, что вы имеете в виду, возможно, вы что-то неправильно поняли и ваш единый язык требует доработки. Давайте названия всему, что вам встречается, и если вы не можете придумать хорошее говорящее название, остановитесь на, возможно, не самом удачном, но подходящем, до тех пор, пока не придумаете что-то более точное.

Предметная модель должна отражать намерения бизнеса. С особым вниманием подходите к именованию методов и свойств классов. Попробуйте описать варианты поведения с использованием единого языка. Не оставляйте места для домыслов в своем коде. Помогайте себе и другим разработчикам, создавая код, помогающий понять предметную область, используя богатство языка предметной области.

Сначала решение задачи, и только потом ее реализация

Программист — это в первую очередь человек, решающий задачи, который использует технические знания и навыки для реализации решения. Хорошие разработчики способны самостоятельно изучать новые технологии и методологии проектирования; однако без способности декомпозиции задач на составляющие и отделения важного от второстепенного хороший разработчик так и останется просто хорошим разработчиком. Вы должны тратить на предметное пространство столько же времени, сколько тратите его на пространство решений.

Одновременно с развитием модели в последовательности итераций предметное пространство тоже должно очищаться, чтобы более явственно отражать истинные намерения заинтересованных лиц. Умение слушать заинтересованных лиц и делать выводы — почему, что и когда — не менее важный навык для разработчиков, который они должны развивать точно так же, как развивают свои навыки в создании программного кода.

Программный код — это продукт DDD, а не процесс; порой задачу можно решить, не имея технического решения.

Не решайте все задачи

Не все задачи одинаково важны; некоторые задачи сложны, но не имеют большого значения для предприятия, поэтому нет смысла тратить силы на поиск автоматизированного их решения. Сложные исключительные ситуации не всегда требуют

автоматизированных решений, зачастую с ними лучше справляются люди. Если задача сложна и представляет исключительную ситуацию, поговорите с заинтересованными лицами и специалистами о ценности ее автоматизации. Возможно, с этой задачей человек справится лучше, а ваши усилия предпочтительнее направить на решение других задач.

Как узнать, что все делается правильно?

Овладев приемами и навыками DDD, вы не получите никаких подтверждающих сертификатов, как, например, в случае с методологией гибкой разработки Scrum, предполагающей обучение и сертификацию. Вашей главной наградой станет продукт, легко понятный, простой в сопровождении, соответствующий ожиданиям заинтересованных лиц, работа с которым доставляет удовольствие.

Также вы увидите, что члены вашей команды лучше стали понимать работу предприятия, свободнее говорить с заинтересованными лицами и способны предлагать решения, о которых заинтересованные лица не догадывались или вообще считали их невозможными.

Погружение разработчиков в предметную область предприятия способствует пониманию всеми в организации истинной ценности бизнеса. Разработчики больше не будут тратить время на усложненные, сугубо технические, необъективные решения с одинаковой архитектурой и требующие одинаковых затрат времени, стремясь получить идеальный код даже в областях, не важных для предприятия. Они научатся раскладывать задачи на составляющие, с помощью специалистов определять наиболее важные области и тратить основное время на них, предоставляя достаточно хорошие и простые решения для всех остальных неспециализированных областей и областей поддержки. Они научатся понимать, в чем заключается истинная ценность, и концентрироваться на ней.

Разработчики будут с большим пониманием концентрироваться на предметной области, а не на исключительно техническом решении. Основное свое внимание они будут уделять ответам на вопросы «что», «зачем» и «когда», оставляя «как» на потом.

Не стремитесь к идеалу

Разработчики, принявшие на вооружение методологию DDD, больше внимания уделяют изучению общей картины, выясняя, куда направить основные усилия. Они используют разные архитектуры для разных частей решения и не стремятся к идеалу в областях, не имеющих большой ценности. Они отдают предпочтение изолированному и действующему программному коду, отказываясь от изящества и украшательств.

Изящество нужно только в реализации смыслового ядра, что объясняется его сложностью или важностью. Это не значит, что остальной код можно писать кое-как, но он должен быть изолирован в четко определенных границах и реализовывать поведение для поддержки смыслового ядра.

Практика, практика и еще раз практика

Разработка программного обеспечения — это процесс обучения и поэтому хорошо соответствует философии DDD. Чтобы достичь вершин, в любом деле нужна практика, практика и еще раз практика. Если вы хотите стать не просто хорошим, а искусным разработчиком, вы должны испытывать страсть к задачам и поиску их решений. Для применения принципов DDD нужна целеустремленная и сплоченная команда — команда, полная решимости овладеть всеми тонкостями ремесла и предметной области, в которой она работает. Страсть есть в каждом из нас, и если вы чувствуете ценность применения DDD, тогда вам остается только вдохновить свою команду и стать пропагандистом. Страсть, как известно, заразна; если вы будете стремиться проводить время со специалистами, чтобы глубже понять предметную область, и сможете показать, насколько благотворно это сказывается на выразительности кода, ваша команда последует за вами.

Многие разработчики теряют интерес, попадая в более зрелое окружение, в связи с боязнью кода, написанного другими программистами. При работе с корпоративными системами рано или поздно наступает момент, когда приходится интегрироваться или работать с существующим окружением. Искусные разработчики превосходно справляются с добавлением новых функций в имеющийся код, делая это технично и безопасно.

Ключевые идеи

- Не нужно считать DDD панацеей от всех бед. Сосредоточьтесь на соответствии потребностям предприятия и изучении предметной области, для которой пишете программное обеспечение.
- Применяйте принципы DDD, только когда это действительно необходимо. Не используйте их как универсальное средство решения всех задач.
- Разделите предметное пространство и сосредоточьтесь основное внимание на смысловом ядре. Все самые интересные и содержательные диалоги состоятся при обсуждении этой области. Именно в этой области вы должны применять принципы DDD и прикладывать основные усилия для увеличения ценности продукта.
- Перед моделированием решения исследуйте действительное положение вещей, а также другие модели и контексты, участвующие в процессе разработки. Кто за них отвечает? Какие отношения будут связывать вас с ними? Какие данные должны передаваться и как?
- Создавайте модели, удовлетворяющие конкретным сценариям. Начинайте с самых рискованных или сложных задач. Привлекайте специалистов в предметной области, но не обременяйте их обсуждением простых операций управления данными.
- Работая с унаследованным окружением, защищайте себя от внешнего кода, не доверяйте никому и надежно охраняйте свои границы. Ограничьте область

для добавления новой функциональности. Не пытайтесь наводить порядок повсюду.

- Постоянно интегрируйте, очищайте и пересматривайте свои модели. Не останавливайтесь, найдя первую рабочую идею. Исследуйте и экспериментируйте, проверяйте другие идеи, создавая новые модели и решения. Спроектируйте хотя бы три рабочие модели.
- Не оставляйте места для домыслов, стремитесь к простоте, откладывая принятие серьезных архитектурных решений и ждите, пока не появятся сложности или новые требования, которые поставят под сомнение ваше решение. Выполняйте реорганизацию с применением стратегических шаблонов, только когда это действительно необходимо.
- Моделирование — это вид деятельности, участие в котором должна принимать вся команда. Моделирование должно применяться каждый раз, когда команда оказывается в тупике, столкнувшись с незнакомой или требующей дополнительного изучения областью. Моделирование не должно ограничиваться определенным этапом в процессе разработки.
- Модель и язык должны развиваться вместе. Модель, которая не поддается простому и непринужденному описанию и обсуждению, получится неполноценной и будет развиваться с большим трудом.

ЧАСТЬ II

Стратегические шаблоны: взаимодействие ограниченных контекстов

Глава 11: Введение в интеграцию ограниченных контекстов

Глава 12: Интеграция посредством обмена сообщениями

Глава 13: Интеграция с RPC и REST посредством HTTP

11

Введение в интеграцию ограниченных контекстов

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Как интегрировать ограниченные контексты, образующие распределенную систему
- Фундаментальные проблемы, сопутствующие созданию распределенных систем
- Как принципы создания сервис-ориентированных архитектур (Service-Oriented Architecture, SOA) помогают в создании слабосвязанных ограниченных контекстов, разрабатываемых независимыми группами разработчиков
- Как благодаря применению реактивной философии DDD выполняются нефункциональные требования с сохранением явной предметной модели, управляемой событиями

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 11 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Следующим шагом после определения ограниченных контекстов в системе (как обсуждалось в главе 6 «Обеспечение целостности моделей предметной области с помощью ограниченных контекстов» и в главе 7 «Карты контекстов») является выбор способов их интеграции для выполнения всех сценариев использования. Одной из самых больших проблем, с которой вы столкнетесь на этом пути, является проектирование надежной распределенной системы. Например, каждый шаг — создание заказа, расчеты с клиентом и организация доставки — может быть выделен в отдельный ограниченный контекст и выполняться независимым программным компонентом, на отдельном компьютере или экземпляре облака. В этой главе вы познакомитесь с основными принципами организации распре-

деленных вычислений, позволяющими сохранить предметные понятия явными и в то же время обеспечить соответствие нефункциональным требованиям, таким как надежность и масштабируемость, свойственным распределенным системам.

Технические проблемы — это лишь один из аспектов интеграции ограниченных контекстов и построения распределенных систем; другим аспектом являются социальные проблемы. Распределенные системы часто слишком велики для одной группы разработчиков и требуют привлечения множества групп, по одной для каждого ограниченного контекста (или нескольких связанных контекстов). Эта глава познакомит вас с шаблонами коллективной работы и организации взаимодействий, которые с успехом используются для создания масштабируемых систем многими командами, практикующими философию DDD. Одним из широко используемых шаблонов является применение сервис-ориентированной архитектуры (SOA).

SOA — это архитектурный стиль создания бизнес-решений в виде слабосвязанных программных служб. Эта глава покажет вам, как, представляя ограниченные контексты в виде служб SOA, можно использовать принципы SOA для создания слабосвязанных ограниченных контекстов и решать технические и социальные проблемы интеграции. Вы также узнаете о преимуществах методологии реактивного, управляемого событиями программирования и о том, как она может использоваться вместе с DDD для моделирования взаимодействий между ограниченными контекстами посредством событий, возникающих в предметной области.

Системы, управляемые событиями, имеют свои проблемы. Наиболее заметными из них являются требования к разработчикам думать иначе при проектировании своих систем, а также необходимость способствовать непротиворечивости в конечном счете (eventual consistency). Поэтому в данной главе также обсуждаются недостатки и варианты борьбы с ними. Кроме того, в этой главе затрагиваются такие оперативные задачи, как мониторинг соглашений об уровне обслуживания (Service Level Agreement, SLA) и обработка ошибок.

После описания в данной главе основных понятий, касающихся интеграции ограниченных контекстов, в следующих главах будут представлены конкретные примеры создания систем, которые интегрируют ограниченные контексты с применением этих понятий. После чтения этой и следующих двух глав вы будете готовы начать применять эти понятия и свои новые навыки для создания распределенных систем, управляемых событиями, совместно с DDD.

Как интегрировать ограниченные контексты

Для успешной работы программные службы должны быть связаны отношениями. Выбор отношений и способов взаимодействий зависит от вас. Это ответственный выбор, так как в значительной мере определяет скорость обмена информацией, эффективность и успешность проекта. Возможно, вам придется интегрироваться с внешним оператором платежей или, может быть, организовать взаимодействия с системой, написанной другой группой разработчиков в вашей компании. Фактически у вас может быть множество внутренних и внешних связей, что весьма характерно для большинства проектов.

Выбирая виды отношений между службами, отражающими организацию предметной области, вы автоматически получаете знакомые вам выгоды DDD, такие как явная модель, облегчающие общение со специалистами в предметной области и позволяющие безболезненно внедрять новые понятия. Многие организации пришли к выводу, что обоснованный выбор границ и протоколов взаимодействий позволяет каждой группе работать независимо, не препятствуя работе других групп.

Выбор одного только метода взаимодействия может обеспечить обслуживание приложением десятка миллионов пользователей в пиковые периоды или вызвать обвал системы и остановку бизнес-операций в тех же условиях. Выбирать метод взаимодействий часто проще после определения отношений, а определение отношений лучше начинать с определения ограниченных контекстов.

Ограниченные контексты независимы

С развитием системы зависимости начинают оказывать все более негативное влияние. Старайтесь избегать образования зависимостей, если на то нет веских оснований. Зависимость в коде означает, что одна группа может нарушить работу кода, созданного другой группой, или тормозить работу другой группы. Зависимость времени выполнения между подсистемами означает, что одна подсистема не может функционировать без другой.

В архитектуре со слабосвязанными ограниченными контекстами, уменьшающей зависимости между ними, каждый ограниченный контекст может разрабатываться изолированно. Его программный код можно развивать, не опасаясь отрицательных последствий из-за изменений в другом ограниченном контексте, и разработчики не должны будут ждать, пока разработчики других ограниченных контекстов реализуют или изменят ту или иную функциональность.

И наконец, чем слабее связаны ограниченные контексты, тем меньше будет узких мест и тем выше вероятность, что функции, ценные для предприятия, будут создаваться и внедряться быстрее и эффективнее.

Проблемы интеграции ограниченных контекстов на уровне программного кода

В главе 6 вы узнали, что ограниченные контексты представляют отдельные подразделения предприятия, такие как отдел продаж или отдел доставки. Вы можете пройти по коридорам организации, в которой работаете, посмотреть на таблички на дверях и выяснить, какие подразделения входят в состав предприятия. Это отличный способ разделить программную систему в соответствии с организационным делением.

Рассматривая кодировку транспортных компаний, вы сосредоточиваетесь на услугах предприятия, связанных с доставкой. Вам придется трудно, если некоторое понятие, касающееся отдела продаж, будет мешать добавлению новой услуги, связанной с включением новой транспортной компании. Фактически изменение кодировки может нарушить работу отдела продаж. Если прежде вы слышали о принципе

единственной ответственности (Single Responsibility Principle, SRP), его применение будет выглядеть для вас вполне логичным. В DDD принцип единственной ответственности можно использовать для изоляции отдельных услуг предприятия в ограниченных контекстах. Иногда вполне допустимо реализовать ограниченные контексты в виде отдельных модулей/проектов в рамках одного решения.

Множество ограниченных контекстов в решении

После определения ограниченных контекстов полезно напомнить всем участвующим в создании системы, что есть более широкая картина. Помещая все ограниченные контексты в единственное решение или репозиторий с исходными текстами, можно помочь разработчикам увидеть, что за пределами их ограниченного контекста существуют другие миры. Понимание общей картины важно, потому что ограниченные контексты должны объединяться для выполнения всех сценариев использования.

Строго говоря, совершенно не обязательно, чтобы все ограниченные контексты находились в единственном решении (например, в решении Visual Studio) или репозитории с исходными текстами. Иногда это просто невозможно, потому что разные ограниченные контексты могут быть реализованы на разных языках программирования и даже выполняться в разных операционных системах.

Не существует единственно правильного ответа на вопрос, где должен находиться программный код ограниченного контекста. Вы должны взвесить все «за» и «против» и выбрать вариант, лучше подходящий для вас.

Пространства имен или проекты для хранения отдельных ограниченных контекстов

Размещение всех ограниченных контекстов в едином решении увеличивает риск образования зависимостей между ними. Если два ограниченных контекста используют один и тот же код из другого проекта в том же решении, возникает опасность, что один ограниченный контекст нарушит работу другого. Представьте, что в общем проекте с названием `Ecommerce.Common` или похожим имеется класс `User`:

```
public class User
{
    public String Name {get; set;}

    public String Id {get; set;}

    public void UpdateAddress(Address newAddress)
    {
        ...
    }
}
```

Если разработчики ограниченного контекста `Shipping` решат изменить реализацию `UpdateAddress()`, это может нарушить работу ограниченного контекста `Sales`,

который использует старые адреса, хранящиеся в определенном месте или формате. Кроме того, разработчикам, использующим общий код, может потребоваться организовать встречу, чтобы решить, какие изменения внести и когда они смогут привести свой код в соответствие с изменениями. Такого рода зависимости между группами могут замедлить разработку проекта и внести нежелательные политические сценарии. На рис. 11.1 изображено единое решение, содержащее множество независимых ограниченных контекстов.

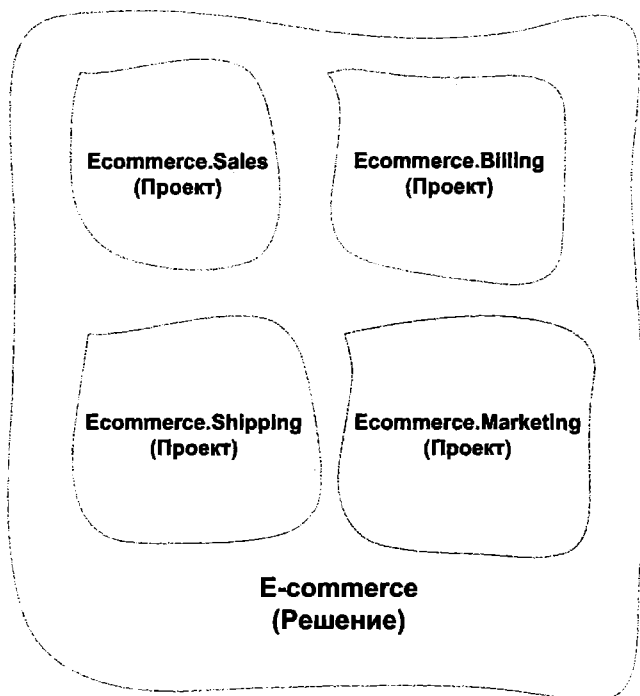


Рис. 11.1. Множество ограниченных контекстов в единственном решении

Интеграция через базу данных

Другой общей зависимостью, замедляющей работу групп разработчиков, является база данных. Например, представьте, что разработчикам ограниченного контекста Sales потребовалось изменить схему User, но никто не уверен, что изменения не нарушат работу ограниченного контекста Shipping или Billing. В этом случае сразу несколько групп будут вынуждены отвлекаться от работы, чтобы привести свои реализации в соответствие с изменениями, необходимыми для ограниченного контекста Sales. Подобные зависимости между группами совершенно нежелательны, потому что замедляют разработку новых функций из-за необходимости согласования действий групп.

Если прежде вам приходилось использовать прием интеграции через базу данных, вам наверняка знакома еще одна типичная проблема, когда все модели, интегри-

рованные через базу данных, имеют внешне схожие предметные понятия. Данный прием превращается в кошмар, когда несколько моделей используют общую схему данных. Например, если говорить о мебели, поставщики могут поставлять мебель массового производства или сделанную на заказ. Эти две услуги наверняка будут иметь множество отличий и почти наверняка будут реализованы в виде двух отдельных ограниченных контекстов. Но если включить их в одно общее решение, велика вероятность, что сходства вызовут желание создать общую схему данных для всех типов мебели, как показано на рис. 11.2.

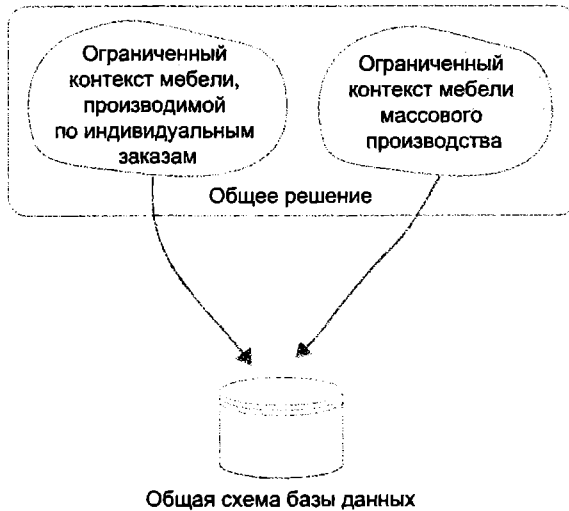


Рис. 11.2. Множество ограниченных контекстов, использующих общую схему

Первое время обе модели ограниченных контекстов — мебели массового производства и производимой по индивидуальным заказам — будут хорошо укладываться в общую схему, но в будущем могут появиться существенные отличия. С появлением различий между моделями столбцы в общей схеме могут перестать соответствовать одной из моделей. Например, модель производства мебели по индивидуальным заказам может потребовать включения нового столбца `manufacturing_priority`, не имеющего смысла для контекста мебели массового производства. Более того, в некоторых ситуациях разные модели могут использовать одни и те же столбцы для разных целей. Отчеты, созданные по данным из такой общей схемы, используемой несколькими ограниченными контекстами, могут содержать ошибки, неоднозначности или вводить работников предприятия в заблуждение.

Обычно нарушение принципа единственной ответственности, когда модели с разной семантикой используют общую схему, обходится очень дорого. Пока объем программного кода невелик, последствия подобного решения не кажутся серьезными. Но с ростом системы и развитием контекстов в разных направлениях проблемы будут накапливаться в геометрической прогрессии.

Участие нескольких групп в разработке общего программного кода

Разделив предметную область на множество ограниченных контекстов с непересекающимся программным кодом, вы ликвидируете целый класс организационных проблем. Когда все разработчики работают с единой коллекцией программного кода, появляется множество незавершенных участков, в отличие от ситуации, когда имеется множество маленьких коллекций (в каждой из которых число незавершенных участков невелико). Большое число незавершенных участков является источником, часто мало заметным, неэффективности в производстве программного обеспечения.

Незавершенные участки становятся проблемой, когда появляется желание выпустить версию с новыми функциями. Как выпустить версию с новой, законченной функцией, если другие функции все еще находятся в разработке? Многие группы используют для этого прием создания отдельных ветвей в исходных текстах; когда работы над некоторой функцией заканчиваются, соответствующая ветвь объединяется с основным стволом и выпускается новая версия. Но это означает появление в проекте долгоживущих ветвей. После двух недель разработки вы можете оказаться так далеко позади основного ствола, что объединение станет в принципе невозможно. Фактически вы потеряете все преимущества непрерывной интеграции. К тому же вы можете потратить на борьбу со слиянием ветвей и выпуск новой версии столько же времени, сколько было потрачено на разработку новой функции.

В больших проектах совсем не редкость, когда 10 или больше разработчиков одновременно работают над разными функциями. Они могут делать большие успехи в добавлении новых возможностей, но, как отмечалось выше, слияние ветвей и выпуск новых версий может оказаться мучительно тяжелым делом. Как следствие, развертывание новых версий будет сопряжено с большим риском, потребуются более тщательный контроль за качеством и больший объем ручного регрессионного тестирования. В особенно тяжелых случаях развертывание новой версии может занять весь рабочий день.

Ныне, когда многие компании используют прием непрерывной поставки и развертывают новые возможности для своих клиентов по несколько раз в день, становится до боли очевидно, что общий программный код, разрабатываемый сразу несколькими группами, обходится слишком дорого для предприятия. Кроме того, работа с единым, монолитным программным кодом не вызывает энтузиазма.

Размывание границ моделей

Если сложная предметная область фактически делится на несколько ограниченных контекстов, но все они реализованы в общем программном коде, границы моделей неизбежно начнут размываться. Код одного ограниченного контекста начнет «зацепляться» за код другого, что приведет к образованию тесных взаимозависимостей. Как отмечалось выше, после образования зависимостей между ограниченными контекстами начнут возникать сложности, препятствующие независимому развитию контекстов и оптимальной работе групп разработчиков.

Однако если для каждого ограниченного контекста создать отдельный проект или модуль, тем самым будет устранена возможность образования тесных связей. Вы можете иметь два ограниченных контекста с похожими реализациями, настолько похожими, что возникает ощущение нарушения принципа «не повторяйся», но обычно это не является проблемой. Очень часто в коде ограниченных контекстов, который выглядит практически одинаковым, неизбежно начинают появляться отличия по разным причинам, таким как появление новых знаний или понятий. Отсутствие связи между ограниченными контекстами избавляет от лишних трений при попытке внедрить в них эти новые понятия или знания.

Но даже если код одинаков и никогда не изменится, дублирование все равно не является проблемой. Дублирование вызывает проблемы, когда имеется единый код и возникает вероятность, что, изменив его в одном месте, вы забудете внести изменения в другом. Однако в случае со слабосвязанными ограниченными контекстами такая проблема возникает крайне редко, потому что ограниченные контексты предполагают выполнение в изоляции друг от друга. Есть очень немного причин, по которым одно и то же понятие в двух ограниченных контекстах должно было бы измениться одновременно. Поэтому не беспокойтесь о дублировании кода, а лучше сосредоточьтесь на изолировании ограниченных контекстов и охране их границ.

Использование физических границ для гарантий чистоты моделей

Для сохранения целостности ограниченных контекстов и гарантий их автономности часто используют архитектуру, не предполагающую наличия общих ресурсов, когда каждый ограниченный контекст получает свой программный код, свое хранилище данных и разрабатывается отдельной группой разработчиков. При правильном применении этот подход позволяет получить системы, построенные из вертикалей, как показано на рис. 11.3.

Когда каждый ограниченный контекст физически изолирован, разработчики одного ограниченного контекста больше не смогут вызывать методы в другом ограниченном контексте или сохранять данные в общую схему. С внедрением физического разделения они вынуждены будут отказаться от приемов, ведущих к образованию тесных связей. Необходимость приложения дополнительных усилий или разъяснения со стороны членов другой группы наверняка остановят их раньше, чем они успеют создать ненужную зависимость.

После изоляции ограниченных контекстов и существенного уменьшения вероятности образования зависимостей другие внешние факторы вряд ли будут оказывать влияние на модель. Например, если в одном ограниченном контексте будет добавлено или уточнено некоторое понятие, это никак не повлияет на похожие понятия в других ограниченных контекстах, что легко могло бы произойти, если бы изменения производились в едином программном коде. По большому счету, явное физическое разделение позволяет развивать каждый ограниченный контекст только под воздействием внутренних причин, благодаря чему получается бескомпромиссная предметная модель, а разработчики получают возможность быстрее и чаще передавать предприятию новые ценные возможности.

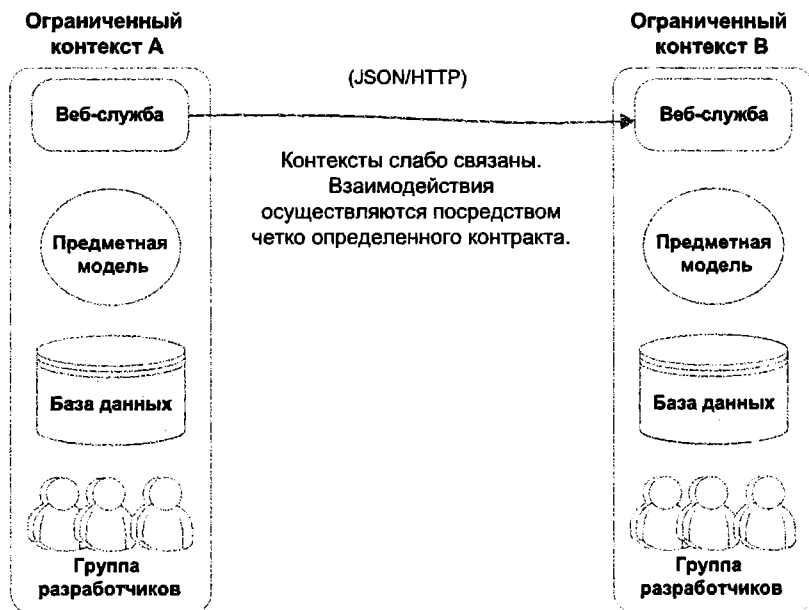


Рис. 11.3. Автономные ограниченные контексты в архитектуре без общих ресурсов

Интеграция с унаследованными системами

При необходимости интеграции ограниченных контекстов, включающих унаследованный код, можно использовать множество шаблонов, ограничивающих влияние унаследованных частей на другие части системы. Эти шаблоны помогут справиться со сложностями и избавят от необходимости ухудшать ясность нового кода, чтобы интегрировать его с унаследованными компонентами.

ПРИМЕЧАНИЕ

Шаблоны, описываемые в этом разделе, были представлены Эриком Эвансом (Eric Evans) в статье «Getting Started with DDD When Surrounded By Legacy Systems», доступной по адресу <http://domainlanguage.com/ddd/strategy/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf>.

Пузырьковый контекст

Группам, не знакомым с философией DDD, но желающим начать применять ее к унаследованной системе, можно посоветовать рассмотреть использование пузырькового контекста (bubble context). Пузырьковый контекст изолирован от общего кода, поэтому он дает чистую основу для создания и развития предметной модели. Помните, что философия DDD работает лучше всего, когда предметная модель находится под полным вашим контролем и вы вольны выполнять итерации с любой скоростью, по мере появления новых знаний в предметной области.

Пузырьковый контекст упрощает частое выполнение итераций, даже когда в работу вовлекается унаследованный код.

Для эффективного применения шаблона пузырькового контекста необходимо реализовать слой преобразования (translation layer), разделяющий пузырьковый контекст и унаследованные модели. Для этого идеально подходит идея предохранительного слоя (Anti-Corruption Layer, ACL) в DDD, как показано на рис. 11.4.

Проектирование и реализация предохранительного слоя (ACL) являются ключевыми аспектами при создании пузырькового контекста. Он необходим, чтобы полностью оградить пузырьковый контекст от особенностей унаследованной системы и в то же время точно преобразовывать команды и запросы пузырькового контекста в команды и запросы унаследованной модели. Он также должен отвечать за преобразование ответов унаследованной системы в формат, понятный пузырьковому контексту. Как следствие, предохранительный слой может получиться сложным, требующим больших трудозатрат.

Автономный пузырьковый контекст

Если требуется интеграция с унаследованным кодом, но нет желания создавать пузырьковый контекст, который слишком сильно зависит от унаследованного кода, возможно использование шаблона автономного пузырькового контекста (autonomous bubble context). В отличие от пузырькового контекста, получающего все данные от унаследованной системы, автономный пузырьковый контекст обладает большей независимостью — он имеет собственное хранилище данных и может выполняться в полной изоляции от унаследованного кода или других ограниченных контекстов, как показано на рис. 11.5.

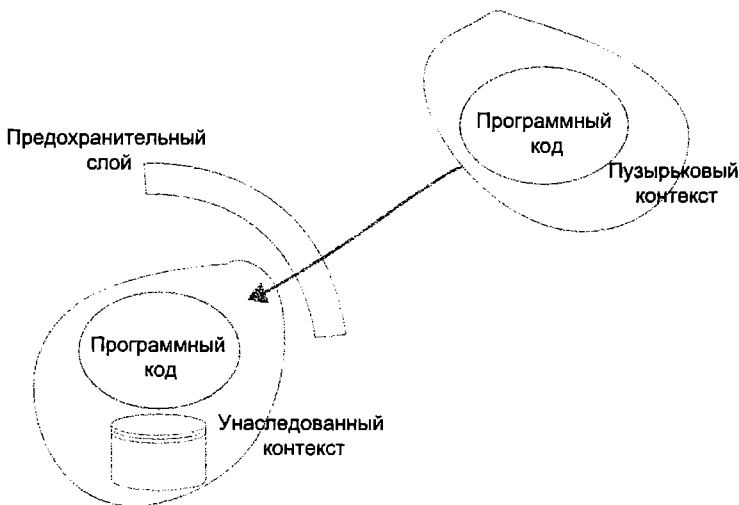


Рис. 11.4. Пузырьковый контекст

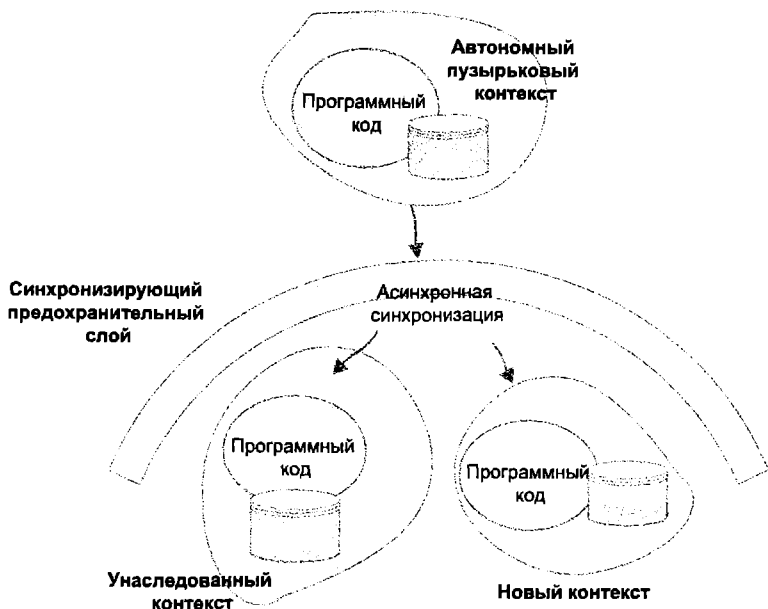


Рис. 11.5. Автономный пузырьковый контекст

Иногда критически важную роль для независимости автономного пузырькового контекста играют асинхронные взаимодействия с другими, новыми и существующими контекстами. Соответственно поддержку асинхронных взаимодействий часто принимает на себя предохранительный слой — синхронизирующий предохранительный слой, как показано на рис. 11.5.

Так как автономный пузырьковый контекст имеет собственное хранилище данных, он не требует вносить изменения в унаследованный программный код или схемы. В хранилище автономного пузырькового контекста можно сохранять любые новые данные. Это важная характеристика, которую следует принимать во внимание при выборе между шаблонами пузырькового и автономного пузырькового контекста. Однако имейте в виду, что цена и сложность асинхронной синхронизации могут оказаться чрезвычайно высокими.

ПРИМЕЧАНИЕ

Далее в этой части книги вы ближе познакомитесь с понятиями асинхронных взаимодействий, включая шаблон шины сообщений (message bus), с примерами на основе протокола HTTP.

Экспортирование унаследованных систем в виде служб

Когда необходимость в использовании унаследованной системы возникает в множестве новых контекстов, цена создания предохранительного слоя для каждого

контекста может оказаться слишком высокой. Вместо этого вы можете использовать прием экспортирования услуг унаследованной системы в виде служб, требующих выполнять меньше преобразований в новых контекстах. Часто проще всего реализовать в унаследованной системе HTTP API, возвращающий данные в формате JSON, как показано на рис. 11.6. Этот шаблон известен под названием «Служба с открытым протоколом» (Open Host Service).

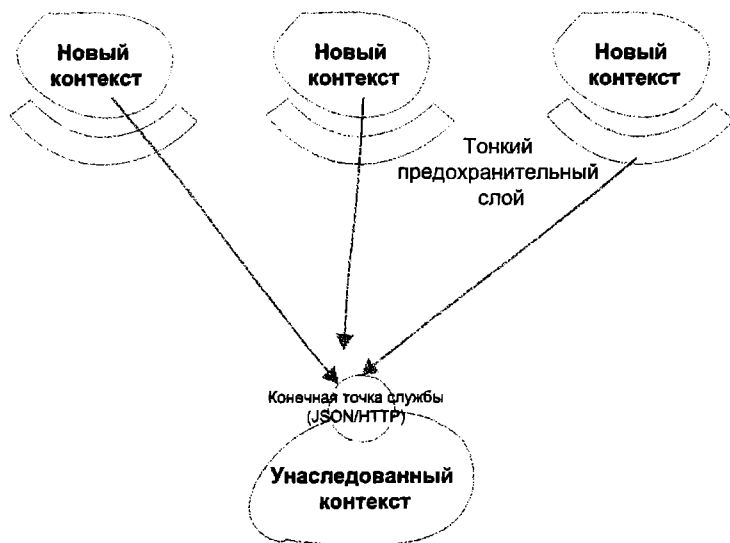


Рис. 11.6. Экспортирование унаследованного контекста в виде веб-службы, поставляющей данные в формате JSON

Каждый контекст-потребитель все еще должен выполнять преобразование во внутреннее представление ответов, полученных от унаследованного контекста. Однако сложность преобразований в этом сценарии существенно снижается за счет простоты API, экспортируемого службой с открытым протоколом.

Несмотря на эффективность приема экспортирования унаследованных контекстов в виде служб при наличии множества потребителей, все же не следует забывать об обеих разновидностях шаблона пузырькового контекста. Экспортирование унаследованных контекстов в виде служб имеет два главных недостатка: во-первых, необходимо внести изменения в унаследованный контекст, что не требуется при использовании шаблонов пузырькового контекста, а во-вторых, порой бывает сложно определить формат представления данных, который легко может обрабатываться всеми контекстами-потребителями.

Следует признать, что выбор стратегии интеграции с унаследованными системами может быть непростым делом. Однако на примере трех подходов, представленных выше, можно видеть, что использование преимуществ DDD при работе с унаследованными контекстами определенно возможно.

Интеграция распределенных ограниченных контекстов

Феноменальный рост числа пользователей Интернета означает, что многие современные приложения должны быть в состоянии пропускать огромные объемы веб-трафика. Если приложение не сможет обслуживать большое число пользователей, предприятие потеряет возможность увеличить свои доходы. Основной проблемой является аппаратное обеспечение: возможностей единственного недорогого сервера обычно недостаточно для обслуживания всех пользователей популярного веб-сайта. Поэтому нагрузка должна распределяться между несколькими машинами. Это относится не только к веб-сайтам, но и ко всем службам, составляющим систему.

Распределение нагрузки между несколькими машинами — настолько распространенная задача, что это привело к буму облачных технологий. Предприятия используют облачные решения для быстрого увеличения ценности своих систем. Если архитектура системы допускает возможность распределения между несколькими машинами, вы можете воспользоваться преимуществами облачного хостинга для эффективного ее масштабирования.

Еще одно важное достоинство распределенных систем — высокая надежность. Если один сервер столкнется с проблемами или выйдет из строя, другие серверы смогут взять на себя дополнительную нагрузку и своевременно обслужить всех пользователей.

Чтобы создать распределенную систему, ее нужно разбить на составные компоненты. Это требование выдвигает на передний план необходимость поддержания ясности моделей, за которую борется философия DDD.

Стратегии интеграции распределенных ограниченных контекстов

Распределенные системы, помогая решить задачу масштабируемости, влекут за собой свои собственные проблемы. К счастью, благодаря наличию множества стратегий интеграции, у вас есть некоторый выбор, определяющий круг проблем, с которыми придется столкнуться. Наиболее распространенными являются вызов удаленных процедур (Remote Procedure Call, RPC) и обмен асинхронными сообщениями (asynchronous messaging), поэтому им будет посвящена большая часть этой главы. Однако иногда неплохой альтернативой могут оказаться общие файлы или базы данных.

Распределенные системы дают ряд нефункциональных преимуществ: масштабируемость, доступность, надежность. *Масштабируемость* (scalability) — это способность выдерживать увеличенные нагрузки, например увеличение числа одновременно обслуживаемых пользователей. *Доступность* (availability) — способность приложения часто подключаться к сети, выполняться и обслуживать своих пользователей. *Надежность* (reliability) — еще одно важное преимущество, имеющее отношение к способности системы справляться с возникающими ошибками. Чуть ниже вы увидите, что все эти преимущества часто находятся в прямой зависимости от тесноты связей между компонентами системы и уровня ее сложности.

При интеграции ограниченных контекстов важно выяснить, какие из нефункциональных преимуществ имеют наибольшее значение для предприятия, чтобы выбрать стратегию интеграции, позволяющую получить эти преимущества с наименьшими усилиями. Некоторые варианты, такие как обмен сообщениями, могут потребовать существенных усилий для их реализации, но они способны обеспечить прочный фундамент для достижения высокой масштабируемости и надежности. С другой стороны, в отсутствие строгих требований к масштабируемости можно организовать интеграцию ограниченных контекстов посредством базы данных, без приложения больших усилий, а затем быстро перейти к реализации более важных особенностей.

К сожалению, нельзя просто спросить у заинтересованных лиц, какая степень масштабируемости или какой уровень надежности системы им нужны. Попробуйте описать им несколько сценариев и указать стоимость каждого из них. Например, можно сообщить, что вы способны гарантировать надежность на уровне 99,9999%, но это будет стоить в три раза дороже, чем гарантии надежности на уровне 99,99%.

Интеграция через базу данных

Простейший способ интеграции ограниченных контекстов заключается в том, чтобы в одном приложении организовать запись информации в определенную базу данных, а в другом — ее чтение. Обычно такой подход выбирается в первой итерации разработки продуктов с минимальным функционалом (Minimum Viable Products, MVP) или для частей системы, не имеющих жестких требований к производительности.

Например, при создании заказа ограниченный контекст **Sales** мог бы добавлять его в таблицу **Sales** в базе данных **SQL**. Позднее ограниченный контекст **Billing** мог бы проверить эту таблицу на наличие новых записей и выполнить сценарий приема платежей для каждой из них.

Реализовать такое решение можно несколькими способами. Например, ограниченный контекст **Billing** может проверять таблицу с определенной частотой, скажем, раз в 5 минут, и отмечать обработанные заказы, изменяя значение столбца в этой же таблице, такого как **paymentProcessed**. Как это делается, можно видеть на рис. 11.7.

Интеграция через базу данных дает еще одно преимущество — слабую зависимость контекстов. Так как обе системы используют для взаимодействий операции записи и чтения с базой данных, реализация каждой из них может развиваться независимо, при условии сохранения совместимости с существующей схемой данных. Однако системы оказываются тесно связанными с общей базой данных, поэтому с ростом системы такое решение может превратиться в головную боль. Блокировки в базе данных — лишь одна из проблем, с которыми вы можете столкнуться. С увеличением числа заказов, добавляемых одной частью системы и изменяемых другой ее частью, два ограниченных контекста начнут конкурировать за обладание ресурсами базы данных. База данных в такой ситуации будет единственным уязвимым звеном (Single Point Of Failure, SPOF) и в случае своего отказа приведет к отказу обоих приложений. Кроме того, некоторые базы данных, такие как **SQL Server**, трудно поддаются переносу на кластерную архитектуру, поэтому придется покупать более мощные и дорогие компьютеры.

Еще один недостаток интеграции через базу данных — отсутствие хороших решений для обработки ошибок. Как быть, если ограниченный контекст **Sales** потерпит аварию до того, как успеет сохранить заказ? Заказ будет потерян? А что, если выйдет из строя сервер базы данных? Означает ли это, что компания не сможет принимать заказы? Эти большие проблемы вам придется решать самостоятельно при использовании приема интеграции через базу данных.

Интеграция через простые файлы

Если в вашем проекте база данных не используется, установка и настройка базы данных только для интеграции двух компонентов может оказаться ненужным излишеством. Это один из примеров, когда интеграция через простые файлы может оказаться неплохим решением. Один компонент записывает некоторую информацию в файлы на сервере, а другой читает эту информацию, подобно тому как это делается в случае интеграции через базу данных. Интеграция через файлы является более гибким решением, чем интеграция через базу данных, но требует больше выдумки, что, в свою очередь, может означать необходимость приложения больших усилий и меньшую скорость выполнения важных для предприятия функций. На рис. 11.8 можно видеть одну из возможных реализаций интеграции через простые файлы, представляющую альтернативу интеграции через базу данных, изображенной на рис. 11.7.

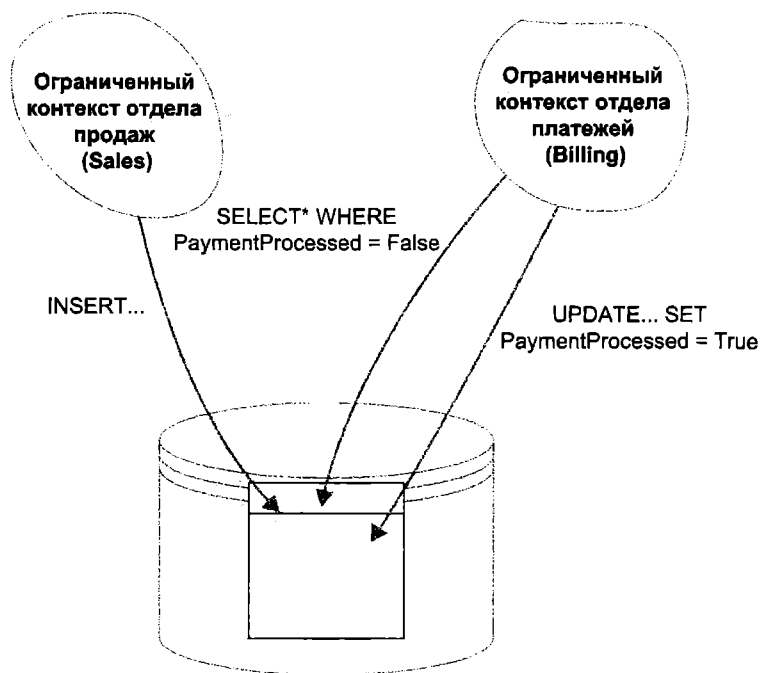


Рис. 11.7. Интеграция через базу данных

Интеграция через файлы обладает тем же преимуществом слабой зависимости между контекстами, что и интеграция через базу данных, но лишена проблемы, порождаемой блокировками в базе данных. К сожалению, в обмен на это работа с файлами потребует от вас приложить больше усилий. Одним из направлений приложения таких дополнительных усилий является выбор формата файлов. Из-за отсутствия схемы данных и стандартного языка запросов вам придется создать собственный формат представления данных и гарантировать его поддержку во всех приложениях. Поскольку это в основном ручной труд, увеличивается вероятность допустить ошибку, хотя это во многом зависит от конкретной ситуации.

Так как интеграция через файлы — это «самодельное» решение, для нее не существует универсальных рекомендаций по увеличению масштабируемости или надежности. Эти характеристики во многом зависят от выбора технологий и их реализации. Если вам требуется по-настоящему масштабируемое решение, реализация интеграции через файлы может оказаться напрасной тратой сил, сопряженной с большим риском. Далее обсуждается прием на основе RPC — обычная альтернатива с относительно неплохими характеристиками масштабируемости и надежности.



Рис. 11.8. Интеграция через простые файлы

RPC

Представьте, что вы могли бы сохранить монолитность реализации, но получить преимущества масштабируемости, характерные для распределенных систем. Именно эта мотивация стоит за использованием RPC. При использовании технологии RPC любой вызов метода может оказаться обращением к другой службе в сети; невозможно утверждать что-то конкретное, не заглянув в реализацию.

Например, сможете ли вы сказать, сколько сетевых вызовов выполняет следующий фрагмент кода?

```
var order = salesBoundedContext.CreateOrder(orderRequest);
var paymentStatus = billingBoundedContext.ProcessPaymentFor(order);
if (paymentStatus.IsSuccessful)
{
    shippingBoundedContext.ArrangeShippingFor(order);
}
```

Глядя на этот фрагмент, нельзя сказать, сколько именно сетевых вызовов он выполняет, потому что неизвестно, что происходит внутри каждого вызываемого метода. Здесь может выполняться и локальный код, и сетевые вызовы других приложений, экспортирующих свою функциональность. Сторонники RPC видят в этом огромную выгоду, потому что это наименее хлопотное решение. В некоторых ситуациях RPC может оказаться лучшим выбором.

Выбирая RPC, вы получаете значительную свободу, потому что сама технология RPC может быть реализована множеством способов. Если бы у вас была возможность побеседовать с разработчиками из разных компаний, применяющих RPC, вы бы нашли примеры использования большинства разновидностей веб-служб — Simple Object Access Protocol (SOAP — простой протокол доступа к объектам), REpresentational State Transfer (REST — передача репрезентативного состояния), eXtensible Markup Language (XML — расширяемый язык разметки), — основанных на разнообразных технологиях, таких как Windows Communication Foundation (WCF — фреймворк для обмена данными между приложениями). Обычно интеграция через RPC реализуется проще, чем через простые файлы, потому что сообщества программистов было разработано множество фреймворков, решающих основные инфраструктурные задачи.

Многие программисты, только начинающие осваивать распределенные системы, испытывают соблазн перед выбором RPC, потому что это дает возможность повторно использовать большую часть их кода. Часто эта особенность оказывается решающим преимуществом RPC, но в ней же кроется недостаток. Далее, в разделе «Проблемы применения DDD в распределенных системах», вы узнаете некоторые подробности о глубоко скрытых недостатках RPC, не позволяющих достигнуть высокой масштабируемости и надежности, из-за которых желательно использовать асинхронные решения и решения на основе обмена сообщениями с обратной связью.

Обмен сообщениями

Как известно, сети ненадежны. Даже такие крупные компании, как Netflix и Amazon, страдают от проблем с сетями, которые приводят к сбоям в системах (<http://www.thewhir.com/web-hostingnews/netflix-outage-caused-by-ec2-downtime-reports>). Реактивные решения¹ помогают справиться с отказами путем увеличения

¹ Решения с обратной связью. — *Примеч. пер.*

надежности за счет организации взаимодействий с применением шаблонов асинхронного обмена сообщениями. Такие решения позволяют системам определить факт потери сообщения и повторить попытку (например, повторно записать сообщение в очередь) или выбрать другой путь.

К сожалению, применение механизмов обмена сообщениями ведет к существенному изменению программного кода, в отличие от RPC. Просматривая код, использующий механизм RPC, почти невозможно догадаться, что он работает с сетью; просматривая код, использующий механизм обмена сообщениями, напротив, сразу видно, что он действует асинхронно и почти наверняка работает с сетью. Но отличия заключаются не только в этом — отличается вся архитектура систем, опирающихся на обмен сообщениями, а разработчики оказываются перед необходимостью освоения сложных механизмов. К счастью, вы познакомитесь с сообщениями в этой и в следующей главах, потому что, наряду с RPC, это один из распространенных вариантов построения распределенных систем с применением методологии DDD. В частности, вы узнаете, что асинхронная природа сообщений также является основой для улучшения масштабируемости.

REST

Если вы хотите получить преимущества масштабируемости и надежности, но вместо механизмов обмена сообщениями использовать протокол передачи гипертекста (Hypertext Transport Protocol, HTTP), попробуйте REST. Архитектура REST предполагает моделирование конечных точек в виде ресурсов в гиперсреде и использование многих преимуществ протокола HTTP, таких как заголовки и методы. С применением REST вы сможете создавать поверх HTTP целые системы, управляемые событиями, пользоваться многими преимуществами систем сообщений и сталкиваться с меньшим числом проблем.

Кроме того, REST является весьма удобным средством экспортирования функциональных возможностей системы в виде прикладного программного интерфейса (Application Programming Interface, API) для интеграции с другими приложениями. После знакомства с основными понятиями распределенных систем в этой главе вы узнаете больше об интеграции с помощью REST в главе 13 «Интеграция с RPC и REST посредством HTTP».

Проблемы применения DDD в распределенных системах

Реализовав ограниченные контексты в виде отдельных служб, взаимодействующих друг с другом по сети, вы получите распределенную систему. Выбор неверной стратегии интеграции в таких системах может приводить к ухудшению производительности и надежности, что, в свою очередь, влечет отрицательные последствия для предприятия. Разработчики должны знать и понимать подходы к построению распределенных систем, чтобы уметь снижать вероятность и серьезность проблем и дать предприятию возможность удовлетворять растущие потребности.

Понимание того, что отказы рано или поздно случатся, равно как и готовность к встрече с ними, являются важнейшими аспектами в строительстве распределенных систем, однако они не присущи RPC.

Проблемы с RPC

Когда приходит время масштабировать приложение и разбить его на множество маленьких подсистем, может появиться соблазн заменить реализацию класса запросом HTTP. Старая логика в этом случае переключается в новую подсистему, которая станет целью этого запроса HTTP. Данное решение выглядит заманчивым, потому что позволяет распределить нагрузку между двумя машинами, не внося существенных изменений в код. Фактически код выглядит как прежде, что демонстрирует предыдущий фрагмент (повторяется ниже):

```
var order = salesBoundedContext.CreateOrder(orderRequest);
var paymentStatus = billingBoundedContext.ProcessPaymentFor(order);
if (paymentStatus.IsSuccessful)
{
    shippingBoundedContext.ArrangeShippingFor(order);
}
```

Каждый вызов метода может обрабатываться локально или выполнять запрос HTTP к другой службе, реализующей фактическую логику. В этом и заключается цель RPC: сделать взаимодействия по сети прозрачными. Взгляните на рис. 11.9, где показана система электронной коммерции, использующая механизм RPC для размещения заказа.



Рис. 11.9. Система электронной коммерции, использующая синхронный механизм RPC

Технология RPC отлично подходит для применения приемов объектно-ориентированного программирования и инкапсуляции, но она имеет существенные недостатки, давно известные сообществу разработчиков распределенных систем. Эти недостатки легко могут свести на нет усилия, потраченные на сотрудничество со специалистами и кропотливую проработку предметных моделей. Чтобы избежать неприятностей, связанных с RPC, далее вам предстоит познакомиться с врожденными проблемами этой технологии, после чего вы узнаете об альтернативных подходах, решающих эти проблемы. Затем вы сможете решить для себя, что лучше подходит для вашего проекта — RPC или обмен сообщениями.

ПРИМЕЧАНИЕ

Строго говоря, это пример синхронного механизма RPC, когда выполнение вызывающего кода блокируется до получения результатов запроса. Тогда имеется возможность реализовать RPC-подобный механизм с применением асинхронных запросов. В этой главе обсуждается только синхронный вариант RPC как более известный. Однако важно знать об отличиях, потому что синхронные взаимодействия являются фундаментальной частью проблемы.

С RPC сложнее обеспечить высокую надежность

Из-за того что технология RPC делает сетевые взаимодействия прозрачными, она помогает напрочь забыть о существовании сети. К сожалению, сетевые ошибки случаются, а это означает, что система, использующая RPC, почти наверняка будет страдать ненадежностью. Сетевые ошибки стали настолько существенным источником проблем для распределенных систем, что им уделено особое внимание в статье «Fallacies of Distributed Computing» (<http://blog.newrelic.com/2011/01/06/the-fallacies-of-distributed-computing-reborn-the-cloudera/>). По сути, о ненадежности сети, ограниченной пропускной способности и задержках говорится так часто, что реализации RPC принимают это как само собой разумеющееся.

В сценарии оформления заказа, где ограниченный контекст Billing выполняет запрос HTTP к сторонней системе приема платежей, если отключится сеть или система приема платежей, заказ нельзя будет оформить до конца. В этот момент потенциальный покупатель может испытать разочарование из-за невозможности приобрести товары, а бизнес испытает еще большее разочарование из-за упущенного дохода. Представьте период рождественских распродаж или большого спортивного события; бизнес-модели многих компаний предусматривают расширение возможностей в такие периоды. Если система отключится, это может иметь серьезные последствия для компании. Далее в этой главе вы узнаете, как избежать этих проблем даже в случае серьезных отказов.

RPC труднее поддается масштабированию

С помощью того же сценария электронной коммерции можно продемонстрировать ограничения масштабируемости систем, использующих RPC. Представьте ситуацию, когда руководитель или владелец предприятия посылает вам электронное письмо, в котором выражает свою озабоченность увеличившимся числом жалоб от пользователей. Например, он может сообщить о сотнях жалоб на слишком

медленную работу веб-сайта или даже на полную невозможность попасть на начальную страницу из-за истечения тайм-аута в браузере. Данная проблема заключается в недостаточном масштабировании, потому что текущая система не справляется с потоком пользователей.

Чтобы удовлетворить пожелания пользователей и уменьшить количество жалоб, веб-сайт должен работать быстрее и иметь возможность одновременно обслуживать большее число пользователей. К сожалению, все ограниченные контексты **Sales**, **Billing** и **Shipping** должны выполнить некоторые операции, прежде чем пользователь получит ответ, как показано на рис. 11.9. Это означает, что для ускорения работы веб-сайта все эти контексты должны получить больше ресурсов. Если перенести на более быстрые серверы только веб-сайт, пользователь все равно вынужден будет тратить то же самое время, ожидая, пока каждый контекст выполнит свою работу. Это не лучшее решение, потому что недостаточно ускорить работу одного только веб-сайта.

Как и прежде, можно воспользоваться альтернативными технологиями, позволяющими масштабировать не только веб-сайт, но и любые другие ограниченные контексты, по отдельности и по мере необходимости. Вы познакомитесь с ними далее в этой главе.

RPC способствует образованию тесных зависимостей

В системах, использующих технологию **RPC**, образуются тесные связи между программными компонентами, что может привести к утрате независимости групп разработчиков и появлению технических проблем, подобных тем, что описывались чуть выше. Когда система выполняет **RPC**-вызов другой системы, возникает множество форм тесных связей, о которых следует знать. Первая зависимость — логическая, потому что логика службы, выполняющей вызов, оказывается зависимой от службы, принимающей вызов. Вторая зависимость — временная, потому что служба, выполняющая вызов, ожидает получить ответ немедленно.

Логическая зависимость

Зависимости между программными компонентами могут возникать даже в отсутствие зависимостей от общего кода. Если служба выполняет вызов другой службы, вызываемая служба должна существовать и действовать в соответствии с ожиданиями вызывающей службы. То есть вызывающая служба оказывается тесно связанной с вызываемой службой. Проблемы, вызываемые логической зависимостью, во многом подобны проблемам зависимостей от общего кода — изменения в одном месте могут нарушить работу в другом. Логические зависимости могут также вызывать болезненные проблемы, когда вызываемая служба оказывается недоступной из-за отказа, потому что в таких ситуациях ни одна из служб не в состоянии продолжить работу. Это вредит надежности.

Временная зависимость

Производительность — важное свойство. Чем выше производительность веб-сайта, тем больше пользователей перейдет в разряд клиентов, оставивших деньги.

Если какая-то часть системы нуждается в ускорении, чтобы в большей мере соответствовать потребностям пользователей, это может оказаться непросто в том случае, когда данный компонент полагается на другой компонент, который тоже вносит свою лепту в продолжительность обработки. Зависимость этого вида называют временной зависимостью, и она присуща технологии RPC. Рисунок 11.9 подчеркивает временную зависимость, показывая, что пользователь вынужден ждать, пока каждый ограниченный контекст выполнит свою часть работы. Это ведет к тому, что компоненты не могут масштабироваться независимо друг от друга.

ПРИМЕЧАНИЕ

Чтобы получить представление о важности производительности, ознакомьтесь с анонсом компании Amazon, в котором заявлено о замеченной корреляции существенного увеличения доходов с увеличением производительности всего на 100 миллисекунд (<http://glinden.blogspot.co.uk/2006/11/marissa-mayer-at-web-20.html>). Компания Google также сообщает о подобных наблюдениях.

Распределенные транзакции ухудшают масштабируемость и надежность

Транзакции — лучший способ поддержания целостности данных. К сожалению, в распределенных системах транзакции обходятся очень дорого, потому что вовлекают в работу сетевые взаимодействия. Это может отрицательно сказываться на масштабируемости и надежности из-за чрезмерно длительного удержания блокировок в базе данных или отказов некоторых компонентов системы. Соответственно вы должны с особой осторожностью относиться к выбору распределенных транзакций, а также RPC, при создании распределенных систем.

Типичным примером распределенной транзакции является планирование поездки в отпуск, которое заключается в бронировании гостиницы и покупке билета на самолет, когда ваша система оформляет бронирование мест в гостиницах, а за билетами обращается к внешней системе. Когда клиент оформляет заказ, система устанавливает блокировку на номер в гостинице в вашей базе данных. Блокировка удерживается до момента, пока не будет куплен билет на самолет. Поскольку выполнение запроса HTTP к авиакомпании через Интернет занимает в среднем несколько секунд, число соединений с базой данных и блокировок в ней растет. По достижении некоторого предела база данных начнет отвергать новые соединения или произойдет что-то похуже. С ростом масштаба системы серьезность этой проблемы будет только увеличиваться, в том смысле, что будут увеличиваться потери возможных доходов.

Еще одной проблемой распределенных транзакций является частичная доступность (точнее, недоступность) некоторых компонентов системы. Представьте, что блокировка в базе данных гостиниц установлена, а сервер системы продажи билетов на самолеты отключился. Для распределенной транзакции этот отказ означает прерывание транзакции и откат к прежнему состоянию. В отсутствие каких-то особых требований к бронированию гостиниц и покупке билетов, которые обычно осуществляются одновременно, это снова ведет к потере доходов из-за преодолимых технических ошибок.

Ограниченные контексты не обязаны быть согласованы друг с другом

Так что же делать, чтобы не использовать блокировки в базе данных и предотвратить нежелательные ситуации, как в примере с бронированием номера, когда билет на самолет невозможно купить? Типичное решение — двигаться дальше, к новому состоянию, исправляющему проблему. В сценарии с планированием отпуска это может означать отмену бронирования гостиницы, если не получилось купить билет на самолет. Но все это случится не в рамках единственной транзакции. Фактически это означает, что ваши ограниченные контексты будут иметь несогласованные представления о мире. Заказ может существовать в одном из них, но не в другом. В другой части системы пользователь может изменить свой адрес, но изменения еще не достигли другого ограниченного контекста. Часто это далеко не идеальное решение, но вы должны помнить, что с увеличением масштаба вам все чаще придется идти на подобного рода компромиссы.

Отказавшись от распределенных транзакций, вы сможете обрабатывать частичные отказы без потери доходов. После бронирования номера в гостинице, к примеру, не важно, отключена система продажи авиабилетов или действует с ошибками. Вы сможете заказать билет, когда эта система станет доступна или возобновит нормальное функционирование. Примеры такой организации вы увидите далее в этой и в следующей главе.

Допущение временных несогласованностей в системе не является радикальным решением. Это вполне распространенный подход, используемый в распределенных системах и известный под названием «потенциальная непротиворечивость» (eventual consistency).

Потенциальная непротиворечивость

Несмотря на то что ваша система может находиться в несогласованных состояниях, она всегда должна стремиться к достижению согласованности (или непротиворечивости) каждого фрагмента данных в некоторый момент времени. (Система в целом, вероятно, никогда не будет находиться в согласованном состоянии.) В предыдущем примере видимая согласованность достигается покупкой билета после восстановления работы системы продаж. По сути, это потенциальная непротиворечивость и ее можно применить к гораздо более широкому кругу сценариев.

ПРИМЕЧАНИЕ

Желающие увидеть более академичное определение и описание потенциальной непротиворечивости (eventual consistency) могут начать свои поиски с Википедии (http://en.wikipedia.org/wiki/Eventual_consistency).

Хочется упомянуть одну важную аббревиатуру, сопровождающую понятие потенциальной непротиворечивости, BASE (происходящую от *англ.* Basically Available, Soft state, Eventual consistency — преимущественно доступный, с изменчивым состоянием, потенциально непротиворечивый). Она является противоположностью

аббревиатуре ACID (Atomicity, Consistency, Isolation, Durability — атомарность, непротиворечивость, изолированность, долговечность), с которой вы наверняка знакомы по опыту работы с реляционными базами данных. Эти аббревиатуры подчеркивают фундаментальные различия в семантике непротиворечивости двух подходов.

В следующей главе вы увидите примеры создания систем обмена сообщениями, использующих потенциальную непротиворечивость так, чтобы не ухудшить впечатления пользователя. Жертвование впечатлениями пользователя может оказаться одним из самых больших недостатков приемов, основанных на потенциальной непротиворечивости. Для многих веб-сайтов типична ситуация, когда пользователю позволяет разместить заказ или добавить некоторую информацию, но нет возможности вернуть подтверждение немедленно. Это отталкивает некоторых пользователей, что в большинстве случаев вполне объяснимо. Но есть также множество историй успеха, начиная от крупных компаний, таких как Amazon (<http://cacm.acm.org/magazines/2009/1/15666-eventuallyconsistent/fulltext>), до совсем небольших. Если вы делаете свою работу, тщательно планируя детали, вы получаете отличный шанс воспользоваться преимуществами масштабирования, которые несет потенциальная непротиворечивость, при этом производя благоприятные впечатления на пользователей.

Помимо изучения примера в следующей главе, вам стоит потратить еще какое-то время на более детальное изучение потенциальной непротиворечивости, прежде чем приступить к созданию действующей системы, использующей это понятие. Можно порекомендовать, например, авторитетную статью Пэта Хелланда: «Life Beyond Distributed Transactions: An Apostate's Opinion» (<http://cs.brown.edu/courses/cs227/archives/2012/papers/weaker/cidr07p15.pdf>). Мартин Фаулер также написал отличную статью, призывающую не бояться жить без транзакций (<http://martinfowler.com/bliki/Transactionless.html>).

Реактивная философия DDD и управление по событиям

За более чем 20 лет специалисты по распределенным системам узнали о множестве ограничений синхронных механизмов RPC и вместо них во многих ситуациях предпочитают использовать решения на основе асинхронных сообщений, управляемых событиями (<http://armstrongonsoftware.blogspot.co.uk/2008/05/road-we-didnt-go-down.html>). Прежде чем начать выяснять, почему этот подход может оказаться более выгодным, мы посмотрим, как с его помощью можно решить проблемы надежности и масштабируемости, выявленные в предыдущем примере с RPC.

В этом разделе вы узнаете, как выбор удачной стратегии интеграции может дать огромные преимущества масштабируемости и надежности, положительно сказывающиеся на предприятии. Здесь будет представлено альтернативное решение, основанное на принципах реактивного программирования (<http://www.reactivemanifesto.org/>) — философии, замещающей RPC асинхронными сообщениями. На рис. 11.10 изображена реактивная система обмена сообщениями, представляющая собой альтернативу реализации RPC, изображенной на рис. 11.9.



Рис. 11.10. Замена RPC реактивным механизмом сообщений

ПРИМЕЧАНИЕ

Приемы асинхронного обмена сообщениями были известны задолго до того, как философия реактивного программирования приобрела популярность. Фактически название «реактивное программирование» было придумано лишь с целью объединения понятия асинхронных сообщений с некоторыми другими идеями, чтобы людям было проще ссылаться на них в своих беседах.

Демонстрация надежности и масштабируемости реактивных решений

На рис. 11.10 можно видеть, что когда пользователь размещает заказ на веб-сайте, в контекст Billing посылается асинхронное сообщение, указывающее, что было произведено размещение заказа. Ограниченный контекст Billing, в свою очередь, генерирует асинхронное событие `PaymentAccepted` после взаимодействий с платежной системой. Ограниченный контекст Shipping, подписанный на событие `PaymentAccepted`, после получения этого события организует доставку заказа. В то же время пользователь, взаимодействующий с веб-сайтом, немедленно получает подтверждение размещения заказа. Позднее, как только все будет подтверждено, он получит еще электронное письмо. Ему не приходится ждать, пока все асинхронные сообщения будут доставлены и обработаны всеми ограниченными контекстами. Асинхронная обработка заказа решает проблему производительности, но решает ли она проблему надежности?

Чтобы решить проблему надежности, каждое сообщение помещается в очередь, где пребывает до тех пор, пока получатель не обработает его. Теперь вернемся к проблемам, которые могут возникнуть в процессе обработки намерения пользователя разместить заказ. Если платежная система окажется неработоспособной, сообщение сохранится в очереди и будет выслано повторно, как только платежная система станет доступна. Если какие-то ограниченные контексты будут испытывать проблемы с аппаратной частью или в них проявятся программные ошибки, сообщения опять же останутся в очереди, ожидая возвращения в строй соответствующих ограниченных контекстов. Как видите, реактивные решения дают платформу для обеспечения исключительной надежности, особенно в сравнении с RPC.

ПРИМЕЧАНИЕ

Большинство фреймворков сообщений облегчают трудности, связанные с хранением сообщений, и принимают на себя все хлопоты по повторной их отправке, как вы узнаете в следующей главе.

Чтобы понять, как реактивные решения увеличивают масштабируемость, вернемся к примеру, в котором пользователи жалуются на непозволительно медленную работу веб-сайта. Вы легко сможете применить оптимальное решение горизонтального масштабирования, добавив еще несколько экземпляров веб-приложения в балансировщик нагрузки, не меняя аппаратное окружение, используемое для ограниченных контекстов. Всякий раз, когда в системе обнаруживается узкое место, вы сможете принимать более точечные решения, куда добавлять новые аппаратные средства. В конечном счете это означает более эффективное (в финансовом смысле) использование аппаратных или облачных ресурсов. В прежнем примере не было никакой возможности реализовать горизонтальное масштабирование каждого из ограниченных контекстов из-за способа их реализации. Это означает доступность только более дорогостоящего решения вертикального масштабирования.

ПРИМЕЧАНИЕ

Под вертикальным масштабированием понимается увеличение производительности за счет улучшения характеристик аппаратных средств, таких как объем ОЗУ или использование более высокопроизводительных серверов. Под горизонтальным масштабированием подразумевается распределение нагрузки между несколькими машинами.

Горизонтальное масштабирование часто обходится дешевле, потому что стоимость добавления новой машины остается прежней. Вертикальное масштабирование, напротив, часто оказывается более дорогостоящим, потому что стоимость более мощных машин может расти скачкообразно.

Проблемы и компромиссы асинхронных сообщений

Будьте осторожны, у вас не должно сложиться впечатление, что реактивное программирование решит все ваши проблемы и сделает жизнь легкой, потому что оно потребует приложить немалые усилия с вашей стороны. Как и многие другие

решения, принимаемые в процессе разработки программного обеспечения, выбор реактивного подхода имеет свои положительные и отрицательные последствия, которые необходимо сопоставлять с ограничениями, имеющимися в настоящий момент. Вот некоторые трудности, с которыми вам наверняка придется столкнуться при создании реактивных приложений: асинхронные решения сложнее в отладке; программный код получается менее прямолинейным, из-за чего другим будет сложнее понять, как он работает; также нельзя списывать со счетов потенциальную непротиворечивость. Кроме того, из-за добавления множества дополнительных инфраструктурных компонентов, осуществляющих доставку и повторную отправку сообщений, возрастет сложность системы.

Однако не нужно бояться необходимости изучения новых приемов. Напротив, это должно захватывать. В следующей главе вы увидите конкретные примеры, как инфраструктурные технологии помогают в отладке асинхронных систем. Вы даже увидите, что при использовании осмысленных имен и следовании некоторым соглашениям по организации программного кода, совсем несложно выяснить, что делает асинхронный код. Также в общих чертах будет обрисована концептуальная структура, которая поможет вам встать на путь к пониманию потенциальной непротиворечивости.

Технология RPC все еще актуальна?

Технологию RPC следует рассматривать как инструмент, который может оказаться лучшим выбором в некоторых ситуациях. Он страдает проблемами с масштабируемостью и надежностью, но иногда они могут быть не самыми существенными ограничениями. В главе 13 демонстрируется, как на основе HTTP создавать службы для систем DDD, использующие RPC так, что их становится возможным применять в ситуациях, описываемых далее.

Короткий период от замысла до воплощения

Иногда бывает необходимо быстро реализовать новую функцию или продукт для оценки клиентами в Интернете. Для предприятия может быть важно победить своих конкурентов или просто поинтересоваться реакцией пользователей на продукт. В таких случаях масштабируемость и надежность не являются проблемой. Поэтому если вы чувствуете, что решение на основе технологии RPC соответствует предъявляемым требованиям, вам придется найти веские причины, чтобы не использовать ее. Одна из причин, почему стоило бы воспользоваться преимуществом короткого периода от замысла до воплощения, состоит в том, что разработчиков, знакомых с HTTP, гораздо больше, чем знакомых с фреймворками обмена сообщениями и сопутствующими концепциями. То есть вы можете быстро сколотить команду и быстро создать систему.

Простота поиска и обучения разработчиков

Трудно представить разработчика, который не был бы знаком с HTTP, еще труднее найти людей, занимавшихся созданием распределенных систем с применением

фреймворков обмена сообщениями. Поэтому в процессе найма новых разработчиков или поиска команды у вас наверняка будет богатый выбор из тех, кому приходилось создавать системы на основе RPC. Если вы поставите целью реализовать систему на основе сообщений, вам может потребоваться обучить людей не только пользоваться фреймворками обмена сообщениями, но и понимать основные идеи, с которыми вы познакомились в этой главе, с тем чтобы они умели проектировать и строить системы на основе сообщений должным образом.

Слабая связь между платформами

Большим недостатком многих фреймворков обмена сообщениями является отсутствие тесной интеграции с разными средами разработки и операционными системами. В следующей главе вы познакомитесь с фреймворком NServiceBus, который прекрасно подходит для интеграции с приложениями .NET, но стоит вам внедрить другой тип шины сообщений, даже созданный специально для .NET, начинают возникать проблемы (хотя и разрешимые). Разработчики некоторых фреймворков обмена сообщениями утверждают, что предусмотрели возможность межплатформенной интеграции. Такие фреймворки имеет смысл исследовать подробнее, если у вас ограниченные контексты выполняются на разных платформах.

ПРИМЕЧАНИЕ

В главе 13 также рассказывается о возможности использовать архитектуру REST, управляемую событиями, если требуется иметь преимущества масштабируемости систем на основе сообщений и использовать протокол HTTP.

Интеграция с внешними системами

В примере с сайтом электронной коммерции была показана необходимость взаимодействий с внешними службами, такими как системы приема платежей. По ряду причин в подобных взаимодействиях через Интернет не используются фреймворки обмена сообщениями. Вместо них используется протокол HTTP. Вам может понадобиться интегрировать свою систему с другими веб-сайтами и приложениями, например, чтобы рекламировать свой продукт и услуги на этих веб-сайтах. Для этого вам почти наверняка придется организовать доступ к REST или RPC API посредством HTTP. Благодаря этому каждый сможет интегрироваться с вашими API, потому что все знают протокол HTTP и умеют работать с ним.

Реактивная философия DDD и SOA

В предыдущем разделе была показана необходимость применения приемов реактивного программирования в сценариях, когда требуется создать масштабируемую и отказоустойчивую распределенную систему. В этом разделе вы узнаете, как разработать структурированное реактивное решение на основе ограниченных контекстов с применением принципов SOA. В процессе вы узнаете, как примене-

ние архитектуры SOA способствуют независимости групп разработчиков, которые могут разрабатывать свои решения параллельно и при этом достигать гладкой интеграции во время выполнения.

Основной причиной использования архитектуры SOA является необходимость изолировать программные компоненты, представляющие разные деловые функции предприятия, такие как расчеты с клиентами или организация доставки. В истинном смысле SOA службы должны быть слабосвязанными с другими службами и в значительной степени автономными — способными выполнять свои функции без помощи других служб. Слабосвязанные службы несут множество технических и деловых преимуществ. Прежде всего группы, разрабатывающие их, могут работать параллельно с минимальными трениями между собой. Далее в этом разделе вы увидите, как этого добиться, но перед этим вы узнаете, почему слабосвязанные службы считаются идеальным шагом от ограниченных контекстов к интегрированным распределенным системам, основанным на реактивных принципах.

ПРИМЕЧАНИЕ

В этом разделе под термином «слабосвязанные» подразумевается минимальная логическая и временная зависимость в сравнении с первоначальным примером RPC. Однако этот термин не имеет фиксированного технического значения в терминологии распределенных систем.

УСТРАНЕНИЕ НЕОДНОЗНАЧНОСТИ SOA

Термин SOA стал довольно неоднозначным в разработке программного обеспечения. Например, Netflix определяет SOA как архитектуру микрослужб (микросервисов) (<http://techblog.netflix.com/2012/06/netflix-operations-part-i-going.html>), тогда как многие компании имеют меньшее число более крупных служб. Также не утихают дискуссии о выборе протоколов взаимодействий. Несмотря на то что часто в реализациях SOA используются шины сообщений, это не является обязательным требованием. Арнон Ротем-Гал-Оз (Arnon Rotem-Gal-Oz) в своей книге подробно исследует, чем является и чем не является архитектура SOA (http://www.manning.com/rotem/SOAp_SampleCh01.pdf).

Важно помнить и понимать, что SOA не означает лишь веб-службы. Сутью SOA является обеспечение слабой связанности для получения технических и коммерческих выгод.

Представление ограниченных контекстов в виде служб SOA

Если принять, что реактивное программирование — это множество низкоуровневых технических принципов, способствующих созданию слабосвязанных программных компонентов, а SOA — высокоуровневая концепция, облегчающая представление слабосвязанных бизнес-функций, тогда их комбинация выглядит

идеальной для создания масштабируемых и надежных распределенных систем для решения коммерческих задач. Остается недостающее пока звено — как объединить все эти достоинства с DDD. Одно из решений — представить ограниченные контексты в виде служб SOA так, чтобы высокоуровневые ограниченные контексты отображались в низкоуровневые программные компоненты, управляемые событиями. Теперь у вас есть все, чтобы соединить достоинства SOA и преимущества масштабируемости/надежности реактивного программирования *reactive programming*. В следующей главе будет показано множество примеров реализации идей, представленных в этом разделе.

Разложение ограниченных контекстов на бизнес-компоненты

Внутри ограниченного контекста может решаться множество задач. Например, в ограниченном контексте *Shipping* (контекст отдела доставки) может присутствовать логика, по-разному обслуживающая срочные и обычные заказы. Выделив каждую большую задачу в отдельный компонент, вы получите ясность в общении с работниками предприятия и все остальные преимущества DDD, которые сопровождают превращение неявного в явное. Кроме того, от создания двух отдельных модулей, каждый из которых решает единственную задачу, увеличится ясность программного кода. Компоненты такого рода называют *бизнес-компонентами* (*business components*). На рис. 11.11 представлено несколько примеров разных бизнес-компонентов, которые могут существовать внутри ограниченного контекста отдела доставки.

Не нужно прилагать все силы, чтобы идентифицировать бизнес-компоненты на ранних этапах. Их можно выявлять с течением времени. Часто идеи возникают

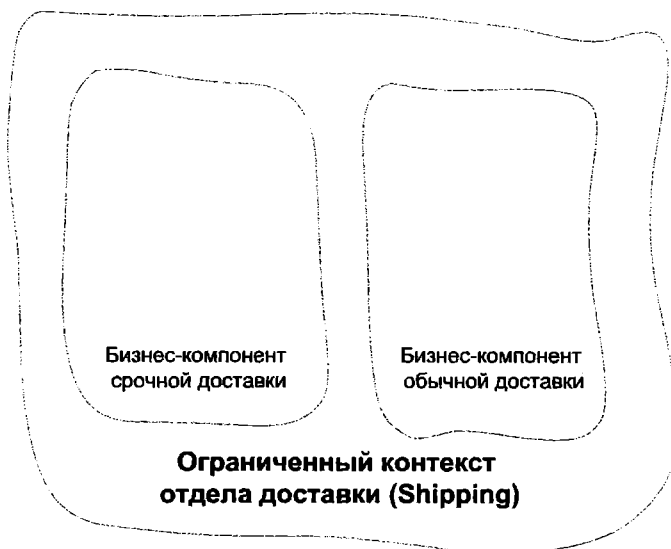


Рис. 11.11. Разложение ограниченного контекста отдела доставки на бизнес-компоненты

во время общения со специалистами или обнаруживаются в коде, который постоянно проверяет одно и то же условие. Все это может служить причиной к началу более глубокого исследования и реструктуризации вашей модели.

Важно помнить, что службы SOA — это лишь логические контейнеры; после разложения ограниченного контекста на бизнес-компоненты в самой службе не должно оставаться артефактов или поведения. Правило, согласно которому службы должны быть отделены друг от друга и не иметь зависимостей, в равной степени применимо к бизнес-компонентам — они не должны зависеть друг от друга, даже если находятся в одном ограниченном контексте. Однако сами бизнес-компоненты — это всего лишь логические контейнеры, вмещающие программные компоненты.

ВНИМАНИЕ

Чтобы получить деловые и технические преимущества в виде возможности независимо разрабатывать слабосвязанные службы, образующие масштабируемые, надежные распределенные системы, бизнес-компоненты не должны выполнять RPC-вызовы друг к другу или обращаться к общей базе данных. С вводом таких точек связи начинают появляться проблемы, обозначенные в начале этой главы.

Опыт и благоразумие помогут вам решить, должны ли вы поддаваться соблазну образования зависимости, который может иметь краткосрочные преимущества.

Разложение бизнес-компонентов на компоненты

Бизнес-компоненты могут отвечать за обработку множества событий, поэтому для большей выгоды их можно разбить на еще более мелкие компоненты. Например, бизнес-компонент `Priority Shipping`, отвечающий за срочную доставку в ограниченном контексте `Shipping` отдела доставки, мог бы обрабатывать сообщения `OrderPlaced` (заказ размещен) и `OrderCancelled` (заказ отменен). Если организовать обработку этих событий в разных программных компонентах, будет проще обеспечить более эффективное распределение аппаратных ресурсов. Предприятию может потребоваться более быстрая реакция на событие `OrderPlaced`, чтобы доставка была организована в максимально сжатые сроки. А скорость обработки событий `OrderCancelled` может оказаться не столь важной из-за небольшого их числа. Поэтому если бизнес-компонент `Priority Shipping` разбить на два компонента — `Arrange Shipping` (организация доставки) и `Cancel Shipping` (отмена доставки), компонент `Arrange Shipping` можно развернуть на очень быстрых, выделенных серверах, а компонент `Cancel Shipping` — на более медленных, виртуальных. На рис. 11.12 показан пример, как бизнес-компоненты в ограниченном контексте `Shipping` можно разбить на программные компоненты.

Самая главная выгода от разложения на компоненты состоит в возможности для предприятия разумнее вкладывать деньги в те части системы, где они наверняка принесут больше отдачи. Но экономия на аппаратных средствах — лишь одна из выгод. Другой выгодой является возможность размещения компонентов в разных



Рис. 11.12. Разложение ограниченного контекста отдела доставки на бизнес-компоненты и компоненты

сетях, в соответствии с приоритетами предприятия и требованиями к скорости работы. Так, некоторые ограниченные контексты, где необходима высокая производительность, могут размещаться на высокопроизводительных серверах и в выделенных сетях с большой пропускной способностью. Вообще говоря, дробление на компоненты помогает получить более высокую гибкость, чтобы точнее соответствовать потребностям предприятия.

Как вы уже, наверное, поняли, компоненты — это единицы развертывания. Это подтверждает рис. 11.13, показывая, как можно развернуть разные компоненты приложения электронной коммерции на разных машинах или экземплярах облака, обладающих аппаратными ресурсами разного уровня (процессор, ОЗУ, накопители SSD и т. д.).

Для ограниченных контекстов и бизнес-компонентов весьма характерно появление проблем надежности или масштабируемости при использовании RPC-вызовов между ними или общей базы данных, однако это не относится к компонентам. Компоненты часто очень тесно взаимодействуют между собой, поэтому использование общих зависимостей, таких как база данных, образующих тесные связи между ними, не обязательно вызывают проблемы. Впрочем, вы все еще можете использовать обмен сообщениями между компонентами, если считаете, что это лучшее решение в данной ситуации.

Рисунок 11.14 должен помочь вам запомнить, что общие зависимости могут существовать только внутри бизнес-компонентов, но не между ними.

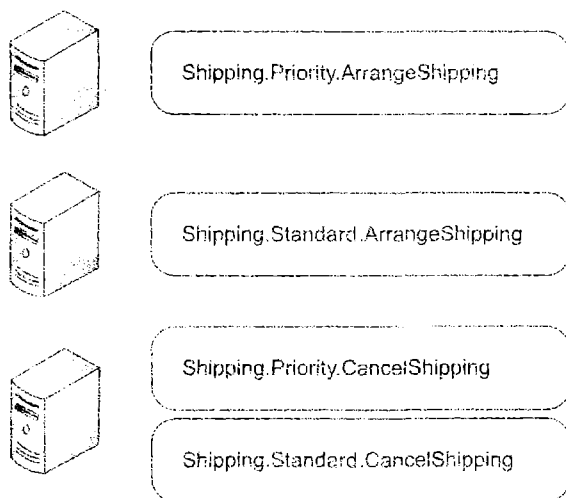


Рис. 11.13. Возможный вариант развертывания компонентов ограниченного контекста на отдел доставки

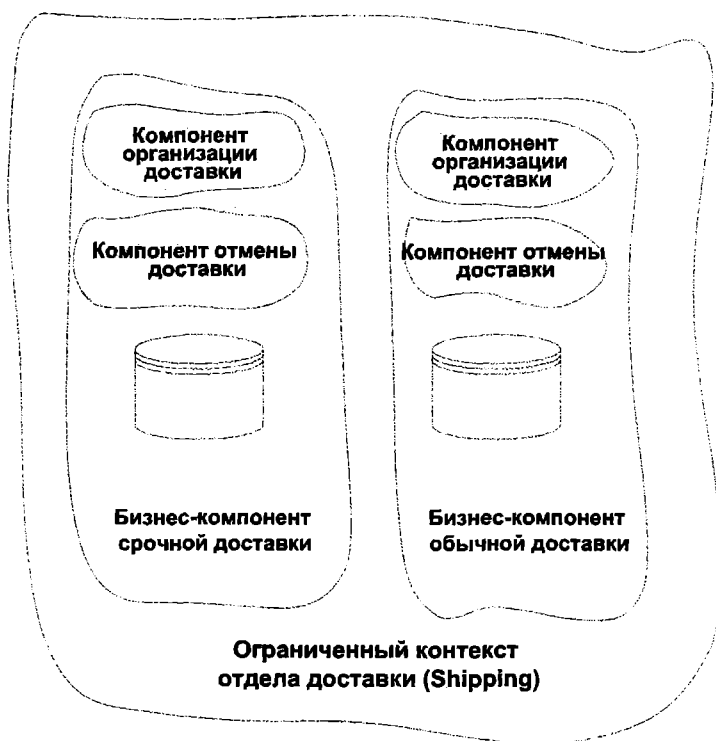


Рис. 11.14. Зависимости допустимы только между компонентами внутри бизнес-компонента

Термин *компонент*, как и многие другие термины в разработке программного обеспечения, носит туманный и двусмысленный характер. В этой главе вам встретились два понятия — бизнес-компоненты и (простые) компоненты. К сожалению, сообществом не было придумано стандартного названия для того, что в этой главе называется простым компонентом. Уди Дахан называет их *автономными компонентами* (autonomous components, <http://www.drdoobbs.com/web-development/business-and-autonomous-components-in-so/192200219>). Другие используют простой термин *компоненты* (components, как в этой главе), опираясь на определение методологии проектирования программного обеспечения на основе компонентных объектов (http://en.wikipedia.org/wiki/Component-based_software_engineering). А третьи предлагают называть их *микрослужбами*, или *микросервисами* (micro services), как рассказывается в следующем разделе.

Еще шаг вперед с архитектурой микрослужб

В Netflix используется прием мелкого дробления служб в SOA, который другие компании начинают применять под названием *архитектура микрослужб* (Micro Service Architecture, MSA). Одними из наиболее выделяемых преимуществ MSA являются короткий период от замысла до воплощения и широкие возможности для экспериментирования. Поэтому если вам доведется сотрудничать с предприятием, часто требующим вносить множество изменений в свою систему и оценивать влияние каждого изменения на развитие новых и существующих возможностей, подумайте о применении MSA.

Следуя рекомендациям в этой главе, можно получить близкую к архитектуре MSA реализацию, в которой каждый компонент будет похож на микрослужбу. Однако MSA имеет ряд существенных отличий. Каждая микрослужба должна быть полностью автономной, чтобы бизнес-функции могли добавляться, удаляться или изменяться без влияния на другие микрослужбы. То есть каждая микрослужба должна иметь собственную базу данных и почти всегда взаимодействовать посредством событий, используя механизм публикации/подписки, потому что команды и RPC вносят взаимозависимость. По крайней мере, именно так был описан стиль MSA Фредом Джорджем, одним из тех, кто впервые начал говорить о MSA в своих лекциях на конференции Oredev 2012. Другой характерной чертой микрослужб, по мнению многих (хотя это довольно спорно), является размер реализации — он не должен превышать тысячи строк кода.

Отличной отправной точкой в изучении архитектуры микрослужб может служить введение от Мартина Фаулера в виде серии статей (<http://martinfowler.com/articles/microservices.html>)¹.

¹ Перевод на русский язык можно найти по адресу <http://habrahabr.ru/post/249183/>. — Примеч. пер.

Ключевые идеи

- В современном мире большинство приложений, которые вам придется создавать, почти наверняка будут распределенными системами, потому что каждый имеющий доступ к Интернету порождает огромный веб-трафик, причем с нескольких устройств.
- Применение положений философии DDD к распределенным системам все еще дает массу выгод, но при интеграции ограниченных контекстов возникают новые проблемы.
- Одни проблемы, такие как масштабируемость и надежность, носят технический характер, а другие, такие как взаимодействия групп и высокая скорость разработки, являются социальными.
- Существует множество приемов создания распределенных систем, помогающих найти золотую середину между простотой и масштабируемостью. Интеграцию через базу данных, например, можно настроить очень быстро, но в окружениях с высокой масштабируемостью использовать это решение не рекомендуется.
- RPC и обмен сообщениями — наиболее распространенные формы интеграции распределенных систем, и вам почти наверняка доведется пользоваться ими. Они имеют существенно отличающуюся природу, поэтому очень важно понимать, какие выгоды и сложности они привнесут в вашу систему.
- В помощь проектированию стратегии интеграции ограниченных контекстов можно использовать философию SOA создания слабосвязанных служб, представляя с ее помощью ограниченные контексты в виде служб SOA.
- Сочетание SOA и реактивного программирования предоставляет платформу для приведения инфраструктуры системы в соответствие с приоритетами предприятия, помогающую решать проблемы надежности и масштабируемости и организовывать группы разработчиков по ограниченным контекстам, чтобы уменьшить потери времени на взаимодействия между ними.

12

Интеграция посредством обмена сообщениями

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Интеграция ограниченных контекстов посредством обмена асинхронными сообщениями с использованием NServiceBus и Mass Transit
- Проектирование и моделирование систем сообщений на основе важных событий в предметной области
- Знакомство с особенностями работы фреймворков обмена сообщениями
- Создание архитектурных диаграмм
- Концептуальные различия между командами и событиями
- Теория и примеры применения стандартных шаблонов организации обмена сообщениями
- Поддержание потенциальной непротиворечивости
- Мониторинг ошибок в системах обмена сообщениями
- Мониторинг соблюдения соглашений об уровне обслуживания (Service Level Agreement, SLA) в системах обмена сообщениями

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 12 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Надеемся, что описание идей в предыдущей главе разожгло в вас интерес к возможностям разработки масштабируемых распределенных систем с сохранением всех выгод предметно-ориентированного проектирования. Цель этой главы — дать вам основополагающие навыки, с помощью которых вы сразу же сможете применять эти идеи на практике. Чтобы получить начальные умения, вы создадите приложение электронной коммерции, использующее механизм обмена со-

общениями на основе фреймворков NServiceBus и Mass Transit. Одновременно вы посмотрите, как внедрять в свои модели знания, полученные от специалистов в предметной области, используя приемы, которые делают предметные понятия в коде более явными. Вы также увидите, что одним из лучших таких приемов является именование сообщений в соответствии с событиями, происходящими в предметной области.

Все примеры в этой главе будут демонстрировать решение типичных задач, часто встречающихся на практике. С некоторыми из них вы уже познакомились в главе 11 «Введение в интеграцию ограниченных контекстов», например с увеличением надежности синхронных HTTP-вызовов служб, не контролируемых вами. Создание систем — лишь малая часть общей картины, поскольку эти системы необходимо также поддерживать и сопровождать. Поэтому в данной главе вы также узнаете, как решать проблемы с изменением форматов сообщений, касающиеся сразу нескольких групп, как контролировать производительность и ошибки в вашей системе обмена сообщениями и как реализовать горизонтальное масштабирование на множестве машин по мере увеличения потребностей предприятия.

Одно из самых больших преимуществ слабосвязанных систем заключается в возможности высокой скорости разработки несколькими группами программистов, как рассказывалось в предыдущей главе. В этой главе вы увидите, как на уровне реализации организовать файлы с исходными текстами и проекты в целях поддержания такой возможности. Но прежде чем опуститься на этот уровень, часто полезно создать общий проект системы, чтобы все группы разработчиков знали, какое место в системе занимают их ограниченные контексты. Поэтому часть данной главы также будет посвящена созданию архитектурных диаграмм, позволяющих наглядно представить важные решения и предметно-ориентированные бизнес-сценарии использования.

Системы обмена сообщениями имеют свои недостатки, и в этой главе будет показано, как обойти некоторые из них. Один из недостатков, в сравнении с вызовами HTTP, заключается в отсутствии стандартных форматов. Это может стать проблемой, если понадобится интегрировать два решения, использующие разные фреймворки обмена сообщениями. Данная проблема частично может быть решена с использованием шаблона обмена сообщениями, получившего название «Мост обмена сообщениями» (messaging bridge). В этой главе вы построите такой мост, соединяющий системы обмена сообщениями NServiceBus и Mass Transit.

Но прежде чем начинать писать код, вам следует познакомиться с некоторыми особенностями систем обмена сообщениями. Они помогут вам лучше понять, на что именно направлены примеры, обсуждаемые далее в этой главе.

Основы обмена сообщениями

Приложения, опирающиеся на обмен сообщениями, в корне отличаются от традиционных нераспределенных объектно-ориентированных приложений. Приложения этого вида не только обладают новыми преимуществами, такими как устойчивость к отказам и масштабируемость, но и содержат проблемные места, такие как

модель асинхронного программирования, требующая совершенно иного мышления. Однако теория, представленная в предыдущей главе, и описание основ обмена сообщениями, о которых рассказывается в этом разделе, должны дать вам все необходимые знания для создания реактивных приложений, использующих асинхронные сообщения. Первое понятие, которое вам нужно освоить, — шина сообщений (message bus). Это связующее звено, обеспечивающее целостность системы.

Шина сообщений

Если в системе имеется единственный централизованный компонент, отвечающий за отправку всех сообщений, вся система может обрушиться, если этот компонент прекратит работу. Не меньшую проблему представляет случай, когда компонент слишком сложен для внедрения в каждую отдельную часть системы, чтобы можно было обеспечить ее масштабируемость в соответствии с потребностями предприятия. Решить эти проблемы можно с помощью *шины сообщений* (message bus), представляющей собой распределенную систему, имеющую агентов в каждом компоненте, который отправляет или принимает сообщения, и позволяющую избежать появления единственной центральной критической точки. Шина сообщений обеспечивает коллективную работу всех частей, как показано на рис. 12.1.

Использование шины сообщений позволяет устранить зависимости между ограниченными контекстами. Каждый ограниченный контекст — точнее, каждый компонент — зависит только от шины. Если компонент выходит из строя, это никак не отражается на других компонентах.

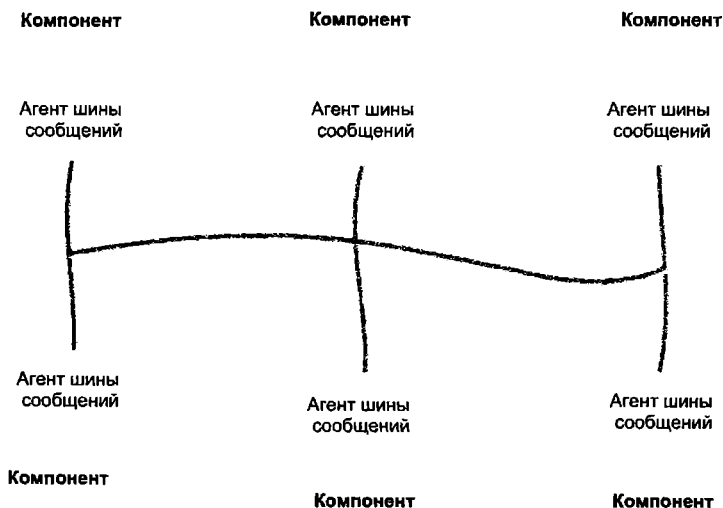


Рис. 12.1. Архитектура шины сообщений

ПРИМЕЧАНИЕ

Централизованный компонент, принимающий и рассылающий все сообщения, называют брокером сообщений (message broker), или просто брокером. Из-за проблем с надежностью и масштабируемостью предпочтение часто отдается шине сообщений. Однако современные брокеры поддерживают горизонтальное масштабирование и многие другие свойства шин сообщений. В качестве примера можно привести пользующийся заслуженной популярностью брокер Apache Kafka (<http://kafka.apache.org/>), созданный и используемый в LinkedIn (<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>). Большинство идей, описываемых в этой главе, также применимо к системам, использующим современные брокеры, такие как Kafka, хотя иногда может потребоваться приложить чуть больше усилий для решения дополнительных проблем.

ПРИМЕЧАНИЕ

Как упоминалось в предыдущей главе, под компонентами в этой главе понимаются отдельные приложения, которые можно распространять и развертывать независимо друг от друга. Вы также можете встретить такие названия, как «автономные компоненты» (autonomous components), «распределенные компоненты» (distributed components), «компоненты, обменивающиеся сообщениями» (messaging components) и даже «микрослужбы» (micro services). К сожалению, сообщество разработчиков пока не нашло точного, однозначного и непротиворечивого названия.

Надежный обмен сообщениями

Отправка сообщений из одного ограниченного контекста в другой может стоить очень дорого в отсутствие гарантий доставки сообщений — клиент будет крайне недоволен, если вы сняли деньги с его кредитной карты, но не смогли организовать доставку купленного им товара из-за потери сообщения где-то в недрах системы. Это одна из причин, объясняющих потребность в надежной доставке сообщений. К сожалению, почти невозможно гарантировать, что сообщение всегда будет доставляться точно один раз. Если сообщение было отправлено, но подтверждение его получения не было принято, оно будет послано повторно. Но что же произойдет, если в действительности сообщение было принято, а подтверждение потерялось? Естественно, то же самое сообщение будет отправлено повторно, исходя из ошибочного предположения, что оно не было принято в первый раз.

Из-за проблем с надежностью доставки было разработано множество шаблонов надежного обмена сообщениями, каждый из которых имеет свои проблемы и компромиссы, в их числе: «доставка не менее одного раза» (at-least-once), «доставка не более одного раза» (at-most-once) и «доставка точно один раз» (only-once).

По мере знакомства с системами обмена сообщениями вы узнаете, что предпочтение обычно отдается шаблону «доставка не менее одного раза» (at-least-once), даже несмотря на риск обработать одно и то же сообщение дважды. Чтобы вы сумели избежать проблем, которые может вызвать такое поведение, например двукратное списание денег со счета клиента за один заказ, в этой главе будет описано, как объединить шаблон «доставка не менее одного раза» (at-least-once) с понятием идемпотентности сообщений.

ПРИМЕЧАНИЕ

Идемпотентными называют сообщения, которые могут посылаться многократно, но обрабатываться только один раз. Например, если сообщение, вызывающее списание денег со счета, посылается дважды, получатель обработает такое сообщение только один раз. Это часто достигается за счет присваивания каждому сообщению некоторого уникального идентификатора, например реквизитов платежа.

Доставка не менее одного раза предполагает повторную отправку сообщений в случае неудачи или отсутствия подтверждения со стороны получателя (из чего можно предположить, что он вышел из строя). Выполнение фактических операций, связанных с повторной отправкой сообщений, возлагается на шаблон «сохранить и передать» (store-and-forward).

Сохранить и передать

Отправленное сообщение может не достигнуть получателя. Например, в случае сетевых неполадок, выхода из строя аппаратного обеспечения или появления программной ошибки. Шаблон «сохранить и передать» (store-and-forward) решает многие подобные проблемы, сохраняя сообщение перед его отправкой. Если сообщение достигло получателя и было подтверждено, его локальная копия удаляется. Но если сообщение не достигло получателя, оно посылается повторно. Далее в этой главе вы увидите, как фреймворки обмена сообщениями принимают на себя большую часть сложностей, позволяя устанавливать правила, которые определяют частоту повторной отправки сообщений и продолжительность ожидания между повторами.

Для хранения сообщений большинство фреймворков использует очереди. Это означает, что, когда Служба А посылает сообщение Службе Б, сообщение будет помещено в очередь Службы Б. Когда Служба Б приступит к обработке сообщения, она извлечет его из очереди. Однако если Служба Б не завершит обработку сообщения, она вновь поместит его в очередь, чтобы повторить попытку позднее.

Команды и события

Иногда сообщения, такие как `PlaceOrder` (заказ размещен) из главы 11, отправляются с целью сообщить о том, что должно произойти. Такого рода сообщения часто называют *командами*. Команды образуют логическую связь между отправителем и получателем, потому что отправитель знает, как получатель должен обработать сообщение. Команды обрабатываются единственным получателем, что лишает систему гибкости. События, такие как `OrderCreated` (заказ создан), напротив, сообщают о происшедшем. События не образуют тесных связей как команды, потому что отправитель не знает, как получатель будет обрабатывать сообщение. Фактически отправитель даже не знает, кто обрабатывает сообщения. Это обусловлено особенностями шаблона «издатель/подписчик» (publish/subscribe).

Более слабая зависимость, обеспечиваемая шаблоном «издатель/подписчик», часто делает события более предпочтительными, чем команды. Одно из существенных преимуществ событий состоит в возможности добавлять новых подпис-

чиков без изменения существующего программного кода. Это позволяет добавлять совершенно новые ограниченные контексты, не меняя существующий код и не замедляя работу других групп разработчиков. В примере сайта электронной коммерции из предыдущей главы можно было бы добавить в поток обработки событий новый ограниченный контекст **Marketing**, представляющий отдел продвижения, просто подписав его на событие **OrderCreated**, как показано на рис. 12.2.

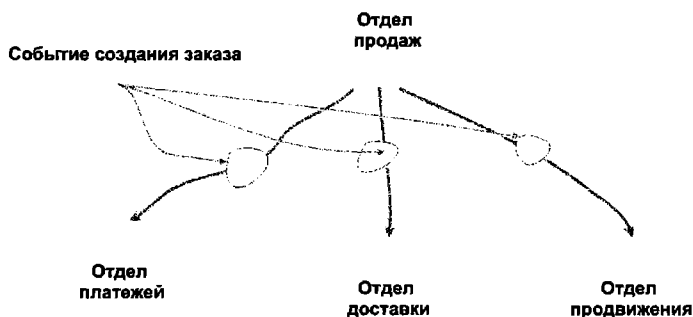


Рис. 12.2. Добавление нового подписчика на получение события не влечет изменения существующего кода

В Сети можно наткнуться на споры о способах именования команд и событий. Вы можете выбрать свой вариант, но по мнению большинства, команды должны получать имена, имеющие вид приказов, определяющие, что должно произойти — **PlaceOrder** (разместить заказ), **ChangeAddress** (изменить адрес), **RefundAccount** (вернуть деньги на счет), — а события должны получать имена, описывающие произошедшее в прошлом времени, — **OrderCreated** (заказ создан), **MovedAddress** (адрес изменен), **AccountRefunded** (деньги возвращены на счет).

Потенциальная непротиворечивость

Реализуя обмен сообщениями в системе, где каждый ограниченный контекст имеет собственную базу данных, есть риск оказаться в ситуации, нетипичной для систем, использующих большие транзакции. Примером может служить выполнение заказа без его оплаты. В системах, использующих транзакции, такое состояние вообще невозможно, потому что создание заказа является составной частью большой, атомарной операции, в рамках которой также осуществляется обработка оплаты. Если в ходе попытки оплатить заказ произойдет ошибка, заказ не будет создан. В решениях на основе сообщений, напротив, такое состояние вполне возможно, потому что они устраняют большие транзакции и являются потенциально непротиворечивыми.

Одним из важнейших аспектов потенциально непротиворечивых систем является создание благоприятного впечатления у пользователей. В непротиворечивых си-

стемах пользователи вынуждены ждать, пока завершится вся транзакция. После чего они точно будут знать, что заказ создан, платеж принят и доставка организована. Однако, как вы узнали в главе 11, большие транзакции не всегда хорошо масштабируются. В системах с потенциальной непротиворечивостью пользователи часто немедленно получают подтверждение, что намерение разместить заказ было принято, а позднее, после обработки платежа и организации доставки, им высылается подтверждение по электронной почте, что на первый взгляд может отрицательно сказаться на впечатлении пользователей.

Вы можете обратить потенциальную непротиворечивость на пользу предприятию, спросив у заинтересованных лиц, что должно происходить в потенциально непротиворечивых состояниях. Иногда таким способом можно вскрыть новые деловые правила или возможности. Для получения дополнительной информации мы настоятельно рекомендуем прочитать статью Уди Дахана «Race Conditions Don't Exist» (<http://www.udidahan.com/2010/08/31/race-conditions-dont-exist/>).

ПРИМЕЧАНИЕ

Большинство шаблонов обмена сообщениями из рассматриваемых в этой главе было формализовано Грегором Хопом и Бобби Вульфом в их весьма авторитетной книге «Enterprise Integration Patterns»¹. Описания всех этих и многих других шаблонов доступны бесплатно на веб-сайте книги (<http://www.eaipatterns.com/>).

Создание приложения электронной коммерции с применением NServiceBus

Теперь у вас есть возможность получить практический опыт создания реактивной системы обмена сообщениями. Сначала вы создадите систему электронной коммерции, с помощью которой пользователи смогут совершать покупки через Интернет. Вы будете использовать команды, события и другие хорошо известные шаблоны, чтобы в полной мере оценить расширенные возможности масштабируемости и надежности, предоставляемые системами обмена сообщениями. Когда вы будете готовы, вашей первой задачей будет загрузить фреймворк NServiceBus, чтобы обеспечить свой компьютер всем необходимым для работы с примерами.

ПРИМЕЧАНИЕ

Чтобы опробовать примеры из этой главы, нужно установить бесплатную версию Visual Studio Express 2013 for Web (<http://www.visualstudio.com/products/visual-studio-express-us>). При желании можно также использовать одну из полных версий Visual Studio.

Установить фреймворк NServiceBus совсем не сложно: просто загрузите и запустите установочный файл. Для работы с примерами в этой главе выбирайте версию 4.3.3

¹ Хоп Г., Вульф Б. Шаблоны интеграции корпоративных приложений. — М.: ООО «И. Д. Вильямс», 2015. — *Примеч. пер.*

(<https://github.com/Particular/NServiceBus/releases/download/4.3.3/Particular.NServiceBus-4.3.3.exe>). Она содержит все необходимое, включая службу очередей сообщений Microsoft Message Queuing (MSMQ) и координатора распределенных транзакций Distributed Transactions Coordinator (DTC). Как вы увидите ниже, вам потребуются также настроить ссылки на сборки (assemblies) NServiceBus в своих проектах.

ПРИМЕЧАНИЕ

Работая с примерами из этой главы, помните, что вы можете задавать любые вопросы или делиться своими мыслями на дискуссионном форуме Wrox.

Проектирование системы

Прежде чем запустить среду разработки, часто полезно графически изобразить, что именно вы собираетесь создать, особенно когда некоторые понятия являются для вас новыми. Визуальное представление того, как разные компоненты укладываются в единую картину, поможет лучше понять, что именно вы в действительности создаете. Проектирование приложения электронной коммерции в этой главе будет выполнено в три этапа. Один шаг — создание диаграммы контейнеров, изображающей группы в приложении, выбранные технологии и протоколы взаимодействий. Другой шаг — создание диаграммы компонентов, отображающей логические связи между ограниченными контекстами. Но первый и самый важный шаг — знакомство с предметной областью.

Предметно-ориентированное проектирование

Как вы узнали в первой части книги, самое важное в разработке программного обеспечения — это, пожалуй, определиться с целью его создания, чтобы суметь спроектировать систему, представляющую ценность для тех, кто будет ее использовать. Поэтому, проектируя систему, большую часть времени следует тратить на создание единого языка и выявление скрытых идей в сотрудничестве со специалистами в предметной области. Некоторые практики DDD рекомендуют начинать с выявления важнейших событий, возникающих в предметной области. Этот этап известен под названием «штурм событий» (Event Storming) и как нельзя лучше подходит для приложений, опирающихся на обмен сообщениями (<http://ziobrando.blogspot.co.uk/2013/11/introducing-event-storming.html>).

События в предметной области

Создавая ограниченные контексты, которые интегрируются посредством общих команд и событий, вы получаете отличную возможность определить, как будут отображаться важные события предметной области в события, возникающие в программной системе. Этот шаблон используется многими известными практиками DDD для явного выражения предметных понятий в создаваемой системе сообщений. Для этого необходимо сначала выявить важные события в предметной области. Их часто называют предметными событиями.

В первой части книги «Принципы и приемы предметно-ориентированного проектирования» было показано множество способов приобретения знаний о предметной области, включая и события. Обычно это происходит во время сеансов переработки знаний совместно со специалистами в предметной области, на неформальных встречах и даже в обеденных перерывах. В предыдущей главе мы видели, что события в предметной области возникают в любом случае, даже в отсутствие программной системы. Вот некоторые из них: «заказ размещен», «оплата получена» и «доставка организована». Этот этап может принести пользу при создании системы событий, если особое внимание уделить подобным предметным событиям.

После того как предметные события выявлены, они становятся частью единого языка (UL) и можно приступить к их объединению, чтобы сформировать законченные сценарии использования. Лучший способ сохранить это знание и поделиться им с другими — отобразить последовательности событий на диаграмме компонентов.

Диаграммы компонентов

Нарисовав схему высокоуровневой логики до начала работы над программным кодом, вы получаете преимущество, позволяющее эффективнее создавать компоненты, потому что вы понимаете, что делаете. И в этом существенную помощь вам окажет диаграмма компонентов. Лучшее время для начала создания диаграмм компонентов — сеансы переработки знаний со специалистами в предметной области. Вы можете вместе набросать основные схемы из прямоугольников и стрелок, отображающие предметные события и процессы. Когда затем вы приступите к программированию, вы уже будете понимать, что именно собираетесь создать, и иметь единый язык с терминологией, необходимой для моделирования системы.

Диаграммы компонентов не имеют формальной структуры, они просто отражают логические связи, или взаимодействия между определенными компонентами. Не следует подниматься на слишком высокий уровень, отображая технологические варианты, точно так же не следует слишком углубляться в детали и показывать имена классов и методов. Пример хорошей диаграммы можно увидеть на рис. 12.3, она показывает потоки сообщений между ограниченными контекстами в будущем приложении электронной коммерции.

Число диаграмм компонентов ограничивается только необходимостью. В данном примере создана единственная диаграмма компонентов, сообщающая порядок размещения заказа. Вы можете взять на вооружение следующее соглашение: диаграмма компонентов должна создаваться для каждого высокоуровневого сценария использования с последующей оценкой ее значимости для вас.

Диаграммы контейнеров

В некоторый момент, после нескольких встреч со специалистами в предметной области, когда начнет появляться понимание предметной области, вы сможете приступить к созданию системы. Вы должны будете реализовать требования предприятия в виде работающей распределенной программной системы, пред-

ставляющей ценность для бизнеса. Сбалансировать функциональные предметные и нефункциональные технические требования вам поможет создание диаграмм контейнеров.

Как будут взаимодействовать разные части приложений? Как проверить, что система обеспечивает необходимый уровень отказоустойчивости и масштабируемости? Как гарантировать понимание всеми разработчиками в группе, что они делают, чтобы они не пошли неправильным путем? Эти вопросы очень важны для большинства программных проектов, а ответить на них вам поможет диаграмма контейнеров. Диаграммы этого вида показывают, как группируются разные части системы, как они взаимодействуют и какие технологии выбраны для их реализации.

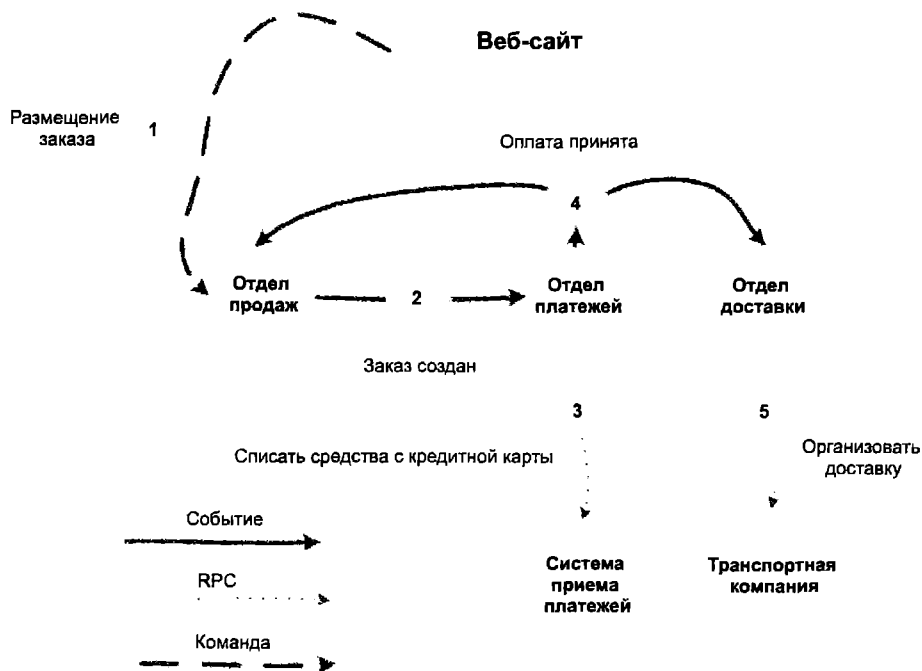


Рис. 12.3. Диаграмма компонентов, отображающая предметные события

ПРИМЕЧАНИЕ

Ключевыми решениями в выборе технологий считаются такие решения, которые оказывают существенное влияние на разработку, распространение или поддержку проекта. Обычно эти решения трудно изменить в последующем. Как правило, к таким ключевым решениям относятся: выбор операционной системы, языка программирования и среды выполнения, веб-серверов, промежуточных программных компонентов и основных прикладных фреймворков, таких как веб-фреймворки или фреймворки поддержки параллельного выполнения.

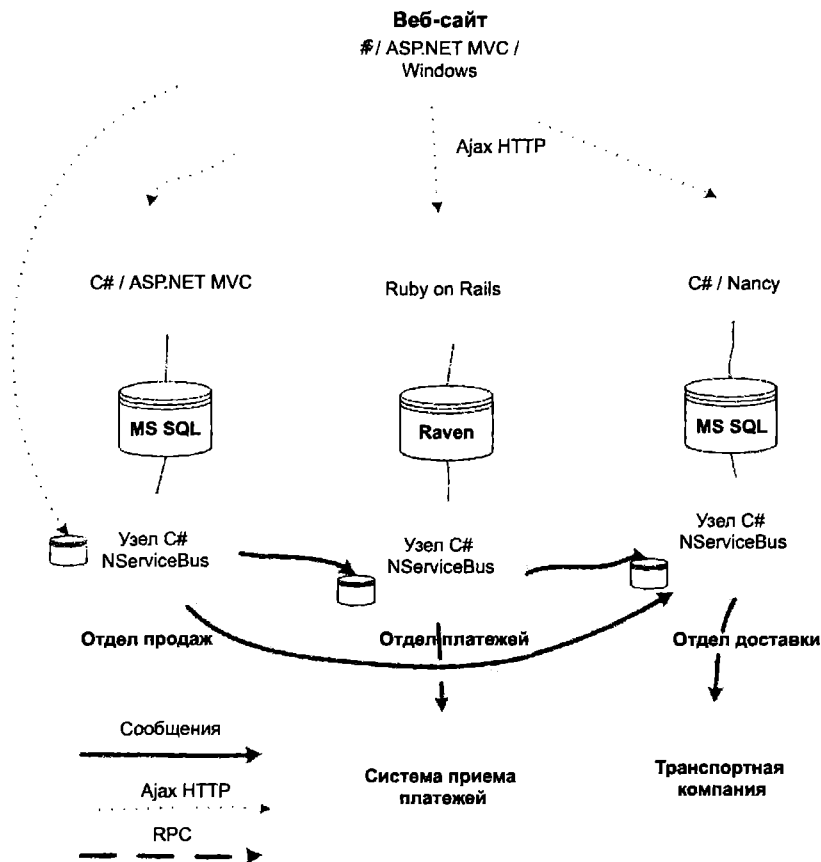


Рис. 12.4. Диаграмма контейнеров типичного приложения электронной коммерции

Взгляните на рис. 12.4, где изображена диаграмма контейнеров приложения электронной коммерции, о котором рассказывается в этой главе.

Отметьте следующие ключевые решения, принятые в ходе проектирования системы:

- Внутренние взаимодействия между ограниченными контекстами осуществляются посредством сообщений.
- Для большей надежности взаимодействия с внешней системой приема платежей выполняются посредством дополнительного шлюза передачи сообщений.
- Каждый ограниченный контекст может использовать разные технологии, включая базы данных.
- Ограниченные контексты не используют общие базы данных и не имеют других зависимостей.

- Веб-сайт извлекает информацию из ограниченных контекстов с использованием прикладных интерфейсов, действующих по протоколу HTTP. Обращения к ним осуществляются посредством запросов AJAX из браузера.

На диаграмме контейнеров (см. рис. 12.4) можно видеть, что проблемы отказоустойчивости и масштабируемости решаются за счет использования применения технологии обмена сообщениями. Здесь также видно, что каждый ограниченный контекст предоставляет свой API, который может использоваться браузером для получения данных по протоколу HTTP и отображения их на странице. Диаграмма также демонстрирует некоторые выбранные технологии. Например, так как каждый ограниченный контекст будет реализован на C#, открывается возможность использовать единую технологию обмена сообщениями (NServiceBus), что упрощает интеграцию. Однако если в требованиях будет указано, что новые ограниченные контексты могут быть написаны на других языках и выполняться на других платформах, вам, возможно, придется задуматься о правильности выбора NServiceBus или любой другой технологии обмена сообщениями и вместо нее отдать предпочтение архитектуре REST (которая будет использоваться для создания системы в следующей главе), чтобы упростить кроссплатформенную интеграцию.

Некоторые проектные решения могут выглядеть очевидными после обсуждения в предыдущей главе. Другие могут показаться странными или даже вызывать несогласие. В этом нет ничего плохого. К концу главы все проектные решения в данном приложении будут рассмотрены в полном объеме. После этого вы сможете точно определиться, согласны вы с ними или нет.

ВНИМАНИЕ

Старайтесь создавать краткие и выразительные диаграммы с одинаковой степенью детализации абстракций. Например, диаграммы контейнеров и компонентов, представленные в этом разделе, можно объединить в одну диаграмму, но она будет содержать слишком много разнородной информации, что усложнит ее понимание. За дополнительной информацией обращайтесь к статье Симона Брауна (Simon Brown) «Coding the Architecture»: <http://static.codingthearchitecture.com/c4.pdf>.

ПРИМЕЧАНИЕ

Не все изображенное на диаграмме контейнеров будет реализовано в этой главе. В частности, в примерах не будут использоваться настоящие базы данных. Они были включены в проект, чтобы более подробно показать процесс проектирования, и подчеркивают тот факт, что ограниченные контексты не обязательно должны использовать одинаковые технологии.

Эволюционная архитектура

Продолжая сотрудничество со специалистами, вы все больше будете узнавать о предметной области. Вы уже знаете, что в процессе обучения вам придется пересматривать и реорганизовывать свои предметные модели и расширять единый язык (UL). Соответственно придется вносить изменения в архитектуру системы,

чтобы отразить в ней новые знания. С появлением новых предметных событий или изменением границ контекстов вам следует попробовать смоделировать архитектуру, опираясь на новые знания, и, соответственно, обновить свои диаграммы. В этом случае ваши диаграммы всегда будут оставаться актуальными, помогать разработчикам принимать обоснованные решения и лучше понимать, что они создают. Кроме того, ваша архитектура будет более точно соответствовать предметной области.

ВНИМАНИЕ

Не забывайте о правильной расстановке приоритетов, развивая архитектуру и пересматривая диаграммы в соответствии с потребностями предприятия. Иногда разработка новой особенности для клиентов экономически выгоднее, чем тщательное соблюдение границ контекстов. Однако не забывайте информировать заинтересованных лиц о цене технических долгов в кратко- и долгосрочной перспективе.

Отправка команд из веб-приложения

Размещение заказа начинается с посещения веб-сайта клиентом. Этот шаг изображен на диаграмме компонентов (см. рис. 12.3) под номером 1. Чтобы начать пользоваться системой сообщений, необходимо создать веб-приложение, способное инициировать процесс отправкой команды `PlaceOrder`, представляющей желание клиента. В данном примере будет использоваться простой веб-сайт MVC 4 с единственной страницей, позволяющей разместить заказ. На рис. 12.5 показано, как выглядит эта страница.

Как видите, внешний вид пользовательского интерфейса не является существенной частью обсуждения в этой главе. Главная наша задача — передать данные в систему обмена сообщениями.



Рис. 12.5. Простая веб-страница для размещения заказа

ПРИМЕЧАНИЕ

Работая с примерами из этой главы, помните, что если вам понадобится помощь или вы почувствуете неуверенность в чем-то, вы всегда сможете получить эту помощь на дискуссионном форуме книги.

Создание веб-приложения для отправки сообщений с помощью NServiceBus

Прежде чем приступить к разработке веб-приложения, создайте в Visual Studio пустое решение с именем DDDesign. В конечном итоге это решение будет вмещать всю систему. На рис. 12.6 изображен процесс создания пустого решения в Visual Studio.

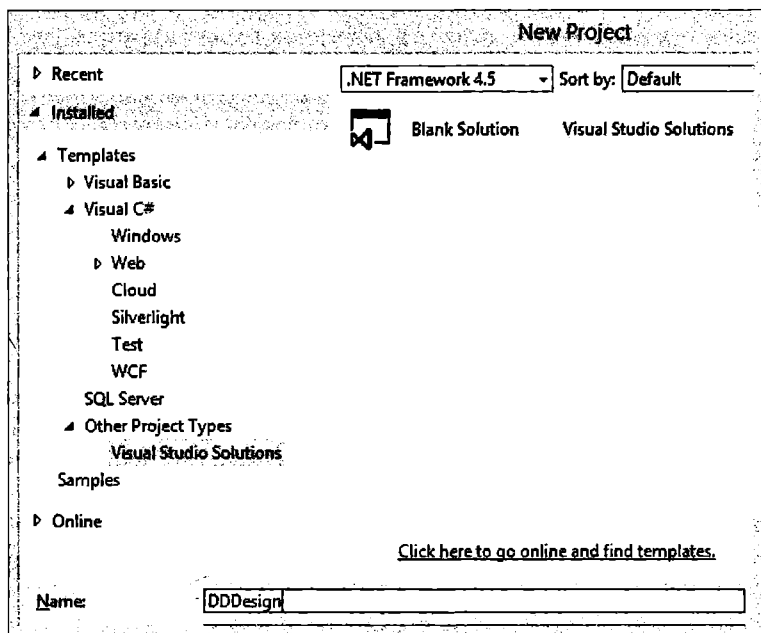


Рис.12.6. Создание пустого решения в Visual Studio

В приложениях, использующих NServiceBus, сообщения представляют собой контракт между ограниченными контекстами. Сообщения должны быть доступны ограниченным контекстам, посылающим их, и ограниченным контекстам, получающим их. Поэтому отличным соглашением является создание для каждого ограниченного контекста проекта, содержащего только сообщения, которые он посылает. Другие ограниченные контексты смогут ссылаться на эти проекты для доступа к сообщениям.

ВНИМАНИЕ

Будьте внимательны: ограниченные контексты должны совместно использовать только проекты с сообщениями. Убедитесь, что это единственная зависимость между ограниченными контекстами в системе обмена сообщениями, потому что общий программный код может привести к образованию тесных зависимостей и лишить выгод, которые несет слабая связанность.

Создание веб-приложения для отправки сообщений с помощью NServiceBus

Прежде чем приступить к разработке веб-приложения, создайте в Visual Studio пустое решение с именем DDDesign. В конечном итоге это решение будет вмещать всю систему. На рис. 12.6 изображен процесс создания пустого решения в Visual Studio.

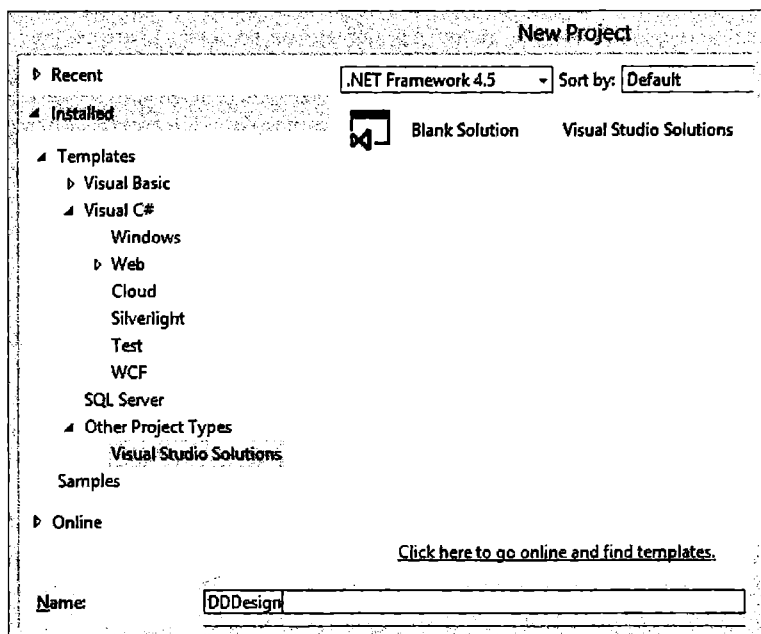


Рис.12.6. Создание пустого решения в Visual Studio

В приложениях, использующих NServiceBus, сообщения представляют собой контракт между ограниченными контекстами. Сообщения должны быть доступны ограниченным контекстам, посылающим их, и ограниченным контекстам, получающим их. Поэтому отличным соглашением является создание для каждого ограниченного контекста проекта, содержащего только сообщения, которые он посылает. Другие ограниченные контексты смогут ссылаться на эти проекты для доступа к сообщениям.

ВНИМАНИЕ

Будьте внимательны: ограниченные контексты должны совместно использовать только проекты с сообщениями. Убедитесь, что это единственная зависимость между ограниченными контекстами в системе обмена сообщениями, потому что общий программный код может привести к образованию тесных зависимостей и лишить выгоды, которую несет слабая связанность.

Определение сообщений

Диаграмма компонентов на рис. 12.3 показывает, что первым шагом является отправка команды `PlaceOrder` ограниченному контексту `Sales` (отдел продаж). Соответственно команда `PlaceOrder` должна принадлежать ограниченному контексту `Sales`. Поэтому в первую очередь нужно создать проект с сообщениями для ограниченного контекста `Sales`. Для этого достаточно добавить в только что созданное решение `DDDesign` новую библиотеку классов C# с именем `Sales.Messages`.

Теперь можно добавить в новый проект `Sales.Messages` команду `PlaceOrder`, представляющую сценарий взаимодействия с пользователем и сообщающую о его желании купить некоторый продукт. Для этого добавьте в корень проекта папку `Commands` и класс с именем `PlaceOrder`:

```
namespace Sales.Messages.Commands
{
    public class PlaceOrder
    {
        public string UserId { get; set; }

        public string[] ProductIds { get; set; }

        public string ShippingTypeId { get; set; }

        public DateTime TimeStamp { get; set; }
    }
}
```

ПРИМЕЧАНИЕ

Вы можете без опасений удалить файл `Class1.cs`, который в Visual Studio добавляется по умолчанию в каждую библиотеку классов C#. И проделывать это всякий раз, создавая библиотеку классов в данной главе.

Теперь можно приступить к созданию веб-сайта, отправляющего команду `PlaceOrder`. Внутри того же решения добавьте новый проект ASP.NET MVC 4, выбрав пустой шаблон и механизм визуализации Razor. Дайте этому проекту имя `DDDesign.Web`. Эти шаги демонстрируются на рис. 12.7 и 12.8.

Далее щелкните правой кнопкой мыши на ярлыке проекта `DDDesign.Web` в окне `Solution Explorer` и выберите в контекстном меню пункт `Add Reference` (Добавить ссылку). Затем выберите пункт `Solution | Projects` (Решение | Проекты). В заключение добавьте ссылку на проект `Sales.Messages`, как показано на рис. 12.9.

Чтобы веб-сайт мог отправлять команды, нужно добавить в него `NServiceBus`. Для этого нужно добавить ссылку на `NServiceBus` с использованием консоли диспетчера пакетов NuGet в Visual Studio. Просто запустите команду `Install-Package`, как показано на рис. 12.10 и в следующем фрагменте:

```
Install-Package NServiceBus -ProjectName DDDesign.Web -Version 4.3.3
```

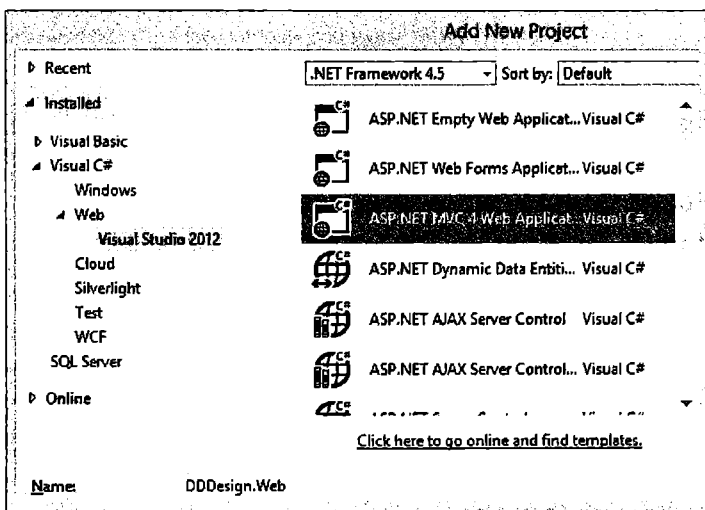


Рис.12.7. Добавление веб-приложения DDDesign.Web типа MVC 4

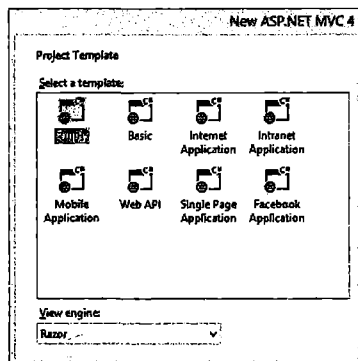


Рис.12.8. Выбор пустого шаблона ASP.NET MVC с механизмом визуализации Razor

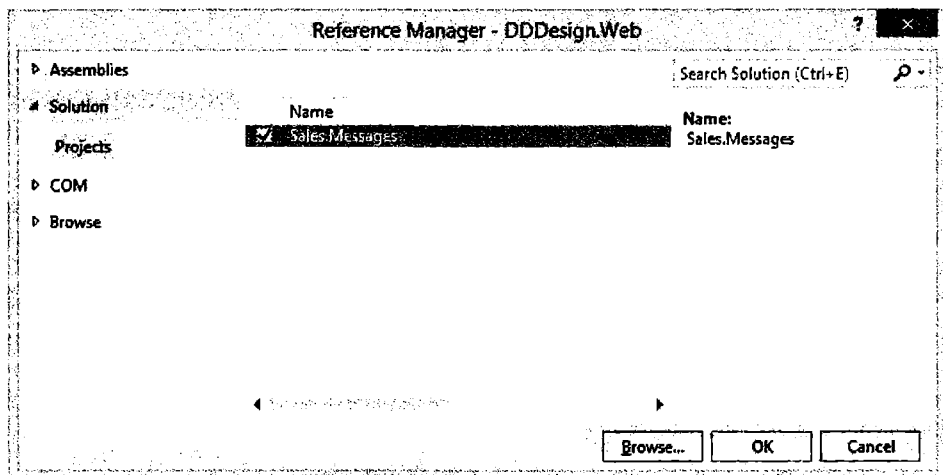


Рис.12.9. Добавление ссылки на проект Sales.Messages в проект DDDesign.Web

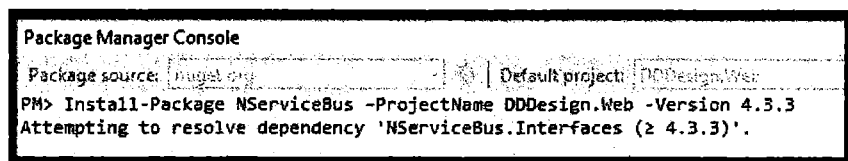


Рис.12.10. Установка NServiceBus с помощью консоли диспетчера пакетов NuGet

Настройка клиента NServiceBus, поддерживающего только отправку сообщений

Теперь, после добавления зависимости от фреймворка NServiceBus, необходимо настроить запуск NServiceBus внутри веб-приложения. Сделать это очень просто благодаря поддержке резидентного хостинга (self-hosting) в NServiceBus, который можно настроить в файле Global.asax.cs, как показано в листинге 12.1.

Листинг 12.1. Настройка клиента NServiceBus только для отправки сообщений внутри ASP.NET MVC 4

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using NServiceBus;
using NServiceBus.Installation.Environments;

namespace DDDesign.Web
{
    public class MvcApplication : System.Web.HttpApplication
    {
        private static IBus bus;

        public static IBus Bus { get { return bus; } }

        protected void Application_Start()
        {
            Configure.Serialization.Xml();
            bus = Configure.With()
                .DefaultBuilder()
                .DefiningCommandsAs(t => t.Namespace != null
                    && t.Namespace.Contains("Commands"))
                .UseTransport<Msmq>()
                .UnicastBus()
                .SendOnly();

            // следующие строки добавляются по умолчанию
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Как видно из листинга 12.1, для настройки клиента не требуется ничего сложного. Однако в этих нескольких строках содержатся важные детали. Во-первых, последняя строка в настройках, `SendOnly()`, сообщает фреймворку `NServiceBus`, что данное приложение будет только отправлять сообщения. Если взглянуть еще раз на диаграмму контейнеров, можно заметить, что это полностью соответствует проекту. В терминологии `NServiceBus` приложения, только отправляющие сообщения, называют *клиентами*.

Метод `DefiningCommandsAs()` используется, чтобы определить соглашение, указывающее фреймворку `NServiceBus`, какие классы в решении являются командами. В данном примере это любой класс, название пространства имен которого содержит слово `Commands`. Если вернуться к определению команды `PlaceOrder`, можно увидеть, что она находится внутри папки с именем `Commands` и, как результат, соответствует данному соглашению. Вы не обязаны следовать этому соглашению; вы можете выбрать любое другое соглашение по своему желанию и даже использовать настройки в формате XML, если возникнет такая необходимость.

Другим ключевым аспектом в настройках является строка `UseTransport<Msmq>()`. Как упоминалось в начале главы, шины сообщений обычно сохраняют сообщения в очередь для большей надежности, чтобы иметь возможность повторно отправлять их в случае ошибки. Эта строка сообщает фреймворку `NServiceBus`, что он должен использовать механизм очередей `MSMQ` компании Microsoft. С тем же успехом можно было бы использовать `RabbitMQ` или `ActiveMQ`.

ПРИМЕЧАНИЕ

За дополнительной информацией о настройках `NServiceBus`, включая выбор соглашения и технологии транспортировки, обращайтесь к официальной документации: <http://particular.net/documentation/nservicebus>.

Отправка команд

Теперь, когда шина сообщений настроена и готова к работе, можно приступить к отправке команд с ее помощью. В этом примере данная операция будет реализована за счет добавления контроллера `OrdersController`, которому веб-страница будет пересылать информацию о заказе пользователя. Контроллер `OrdersController`, получив заказ, будет в свою очередь посылать команду `PlaceOrder`, используя для этого шину, настроенную выше. Чтобы добавить контроллер `OrdersController`, щелкните правой кнопкой мыши на папке `Controllers` в проекте `DDDDesign.Web` и выберите пункт `Add → Controller` (Добавить контроллер). Введите имя `OrdersController` контроллера и щелкните на кнопке `Add` (Добавить), как показано на рис. 12.11.

После создания контроллера `OrdersController` замените содержимое файла программным кодом, представленным в листинге 12.2.

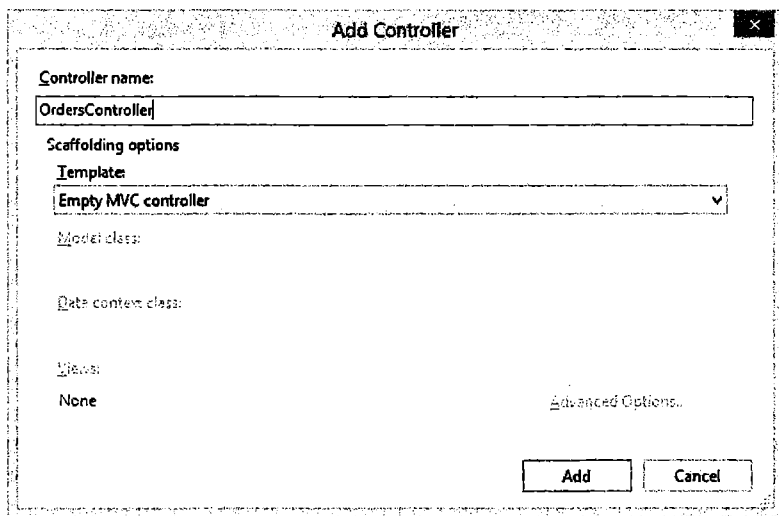


Рис. 12.11. Добавление контроллера заказов

Листинг 12.2. Контроллер `OrdersController`, посылающий команду `PlaceOrder` с помощью `NServiceBus`

```
using Sales.Messages.Commands;  
using System;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;
```

```
namespace DDDesign.Web.Controllers  
{  
    public class OrdersController : Controller  
    {  
        [HttpGet]  
        public ActionResult Index()  
        {  
            return View();  
        }  
  
        [HttpPost]  
        public ActionResult Place(string userId, string productIds,  
                                string shippingTypeId)  
        {  
            var realProductIds = productIds.Split(',');  
            var placeOrderCommand = new PlaceOrder  
            {  
                UserId = userId,  
                ProductIds = realProductIds,  
                ShippingTypeId = shippingTypeId,  
            }  
        }  
    }  
}
```

```
        TimeStamp = DateTime.Now
    };
    MvcApplication.Bus.Send(
        "Sales.Orders.OrderCreated", placeOrderCommand
    );
    return Content(
        "Your order has been placed. " +
        "You will receive email confirmation shortly."
    );
}
}
```

Этот код был преднамеренно упрощен, чтобы подчеркнуть аспекты, связанные с обменом сообщениями. Отправка сообщений с помощью `NServiceBus` выполняется относительно легко. Достаточно просто создать экземпляр сообщения и вместе с именем получателя передать его в вызов метода `Bus.Send()`, который отправит сообщение асинхронным способом. Отправляя команды, убедитесь, что вы не забыли указать получателя, потому что команды обрабатываются только в одном месте. Позднее вы узнаете, что для рассылки общедоступных событий указывать получателя необязательно. Это важное отличие команд, о котором следует помнить.

В данном случае можно также видеть пример потенциальной непротиворечивости (*eventual consistency*). Пользователь немедленно получает ответ, еще до того, как заказ будет успешно обработан всеми компонентами системы. Этот подход подробно описывался в главе 11 как решающий проблему отказоустойчивости на каждом шаге выполнения сценария. Этот же подход позволяет масштабировать веб-сайт независимо от других компонентов системы, потому что избавляет от необходимости ждать завершения обработки, как это потребовалось бы в решении на основе вызовов удаленных процедур (*Remote Procedure Call*, *RPC*). Если преимущества данного подхода вызывают сомнения, будет полезно бегло прочитать предыдущую главу еще раз.

ВНИМАНИЕ

Метод `Place` в листинге 12.2 принимает идентификаторы товаров (`productIds`) в виде строки со списком идентификаторов, разделенных запятыми. Так поступать не рекомендуется. Все веб-фреймворки предлагают более эффективное решение, которое обычно называют *связыванием модели* (*model-binding*). В этом примере специально применено упрощенное решение, чтобы подчеркнуть аспекты, связанные с использованием `NServiceBus` и обменом сообщениями. Это же замечание относится к возвращаемому значению в виде строки. В действующих проектах следует возвращать веб-страницу или ответ `API`.

Единственное, что осталось сделать, чтобы получить возможность действительно отправлять команды `PlaceOrder`, — создать веб-страницу с формой для ввода информации о заказе. Чтобы создать веб-страницу, сначала добавьте папку `Orders` в папку `Views`, присутствующую в проекте `DDesign.Web`. Внутрь вновь созданной

папки `Orders` добавьте файл с именем `Index.cshtml`. Затем удалите все его содержимое и замените программным кодом из листинга 12.3. Если вы все сделаете правильно, ваше решение будет иметь структуру, изображенную на рис. 12.12.

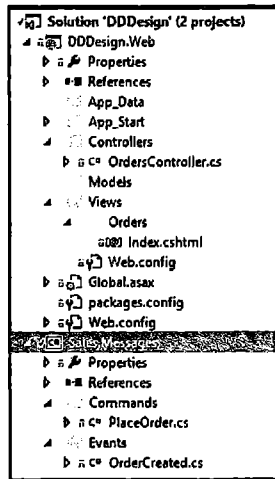


Рис. 12.12. Структура решения после добавления контроллера и представления заказов

ВНИМАНИЕ

На протяжении всей этой главы используются статические классы и статические переменные, потому что они упрощают решение и позволяют демонстрировать примеры, сосредоточенные на обсуждаемых понятиях. При создании действующих приложений следует очень осторожно относиться к статическим методам и классам, потому что они способствуют образованию тесных связей, ухудшают простоту сопровождения кода и его тестирования. Если вы не знакомы с принципом инверсии зависимости (*dependency inversion*), вам определенно стоит почитать о нем. Отличным источником информации по этой теме может служить статья <http://martinfowler.com/articles/dipInTheWild.html>.

Вот некоторые примеры использования статических членов в этой главе, не рекомендуемых для применения в действующих приложениях: `MvcApplication.Bus.Send()`, `Database.GetCardDetailsFor()` и `PaymentProvider.ChargeCreditCard()`.

Листинг 12.3 содержит простую форму HTML, позволяющую ввести информацию о заказе. На рис. 12.5 показано, как выглядит эта страница в окне браузера.

Листинг 12.3. `Index.cshtml`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Place an order</title>
  </head>
  <body>
    <h1>Place an order</h1>
```

```

<form method="post" action="/orders/place">
  <p>
    UserId: <input type="text" name="userId" />
  </p>
  <p>
    ProductIds: <input type="text" name="productIds" />
  </p>
  <p>
    ShippingTypeId: <input type="text" name="shippingTypeId" />
  </p>
  <input type="submit" value="Place order" />
</form>
</body>
</html>

```

Обработка команд и публикация событий

Теперь, когда в веб-приложении имеется страница, позволяющая клиентам создавать заказы, и контроллер, использующий информацию о заказе для отправки команды `PlaceOrder` в шину, необходимо реализовать фактическую обработку команд `PlaceOrder`. Если еще раз взглянуть на диаграмму на рис. 12.3, можно увидеть, что команды `PlaceOrder` обрабатываются ограниченным контекстом `Sales`, который после обработки команды публикует событие `OrderCreated`. Поэтому в данном разделе будет создан компонент в ограниченном контексте `Sales`, содержащий описанную логику.

Создание сервера `NServiceBus` для обработки команд

Серверы `NServiceBus` напоминают службы `Windows` — это приложения, действующие в фоновом режиме и не имеющие пользовательского интерфейса. В данном примере мы добавим сервер `NServiceBus` для обработки команд `PlaceOrder` внутри ограниченного контекста `Sales`. Следуя соглашению об именовании `{BoundedContext}. {BusinessComponent}. {Component}`, добавьте новую библиотеку классов `C#` с именем `Sales.Orders.OrderCreated`. Имя `OrderCreated` указывает на тип сообщений, публикуемых компонентом. Это соглашение об именовании компонентов будет использоваться на протяжении оставшейся части главы.

Чтобы превратить библиотеку классов в сервер `NServiceBus`, нужно установить `NuGet`-пакет `NServiceBus.Host`. Сделать это можно, выполнив следующую команду в консоли диспетчера пакетов `NuGet`, внутри `Visual Studio` (команда должна вводиться в одну строку):

```

Install-Package NServiceBus.Host -ProjectName
Sales.Orders.OrderCreated -Version 4.3.3

```

Серверы `NServiceBus` имеют множество типовых настроек по умолчанию. Но они ничего не знают об используемых у вас соглашениях. Чтобы зафиксировать соглашения в проекте `Sales.Orders.OrderCreated`, замените содержимое автоматически созданного класса `EndpointConfig` кодом из листинга 12.4. Все, что он дела-

ет, — это применяет те же умолчания к местонахождению команд, установленные для веб-приложения, плюс похожее соглашение о местонахождении событий. Обратите особое внимание на два дополнительных интерфейса в цепочке наследования этого класса: `IWantCustomInitialization` и `AsA_Publisher`.

Листинг 12.4. Добавление нестандартных соглашений в класс `EndpointConfig` сервера `NServiceBus`

```
using NServiceBus;
namespace Shipping.BusinessCustomers.ShippingArranged
{
    public class EndpointConfig : IConfigureThisEndpoint, AsA_Server,
                                IWantCustomInitialization, AsA_Publisher
    {
        public void Init()
        {
            Configure.With()
                .DefiningCommandsAs(t => t.Namespace != null
                    && t.Namespace.Contains("Commands"))
                .DefiningEventsAs(t => t.Namespace != null
                    && t.Namespace.Contains("Events"));
        }
    }
}
```

Предполагается, что внутри компонента `Sales.Orders.OrderCreated` будут обрабатываться команды `PlaceOrder`. Чтобы получить такую возможность, нужно добавить ссылку на проект `Sales.Messages`.

Для организации обработки сообщений очень удобным выглядит соглашение, которое заключается в создании класса с именем `{MessageName}Handler`. То есть, согласно этому соглашению, в данном примере следует создать класс `PlaceOrderHandler` в корне проекта `Sales.Orders.OrderCreated`. Когда вы это сделаете, замените содержимое созданного файла программным кодом из листинга 12.5.

Листинг 12.5. Класс `PlaceOrderHandler` для обработки команд `PlaceOrder`

```
using System;
using NServiceBus;
using Sales.Messages.Commands;

namespace Sales.Orders.OrderCreated
{
    public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
    {
        // зависимость, внедряемая фреймворком NServiceBus
        public IBus Bus { get; set; }

        public void Handle(PlaceOrder message)
        {

```

```
        Console.WriteLine(  
            @"Order for Products:{0} with shipping: {1}" +  
            " made by user: {2}",  
            String.Join(", ", message.ProductIds),  
            message.ShippingTypeId, message.UserId  
        );  
    }  
}
```

Как показано в листинге 12.5, класс `PlaceOrderHandler` просто выводит в консоль информацию, полученную в команде `PlaceOrder`. Это временная «заглушка», чтобы можно было запустить приложение и убедиться, что все работает. Однако в коде есть некоторые детали, касающиеся обработчиков сообщений `NServiceBus`, которые следует отметить. Во-первых, `NServiceBus` знает, что любой класс, наследующий `IHandleMessages<T>`, обрабатывает сообщения типа `T`. Во-вторых, фреймворк автоматически внедрит экземпляр `IBus`, если объявить свойство с типом `IBus` и именем `Bus`. Это очень удобно, в чем вы очень скоро убедитесь, потому что позволяет посылать другие сообщения, включая события прямо из обработчиков сообщений.

Настройка решения для тестирования и отладки

Прежде чем перейти к обсуждению публикации событий, самое время проверить работоспособность того, что уже создано, и получить общее представление о том, как работает механизм обмена сообщениями. `NServiceBus` помогает упростить отладку распределенных систем на локальном компьютере. Все, что вам потребуется, — это настроить запуск каждого проекта, как описывается ниже.

1. Щелкните правой кнопкой мыши на файле решения в панели **Solution Explorer** (Обозреватель решения) и выберите пункт **Properties** (Свойства), как показано на рис. 12.13.
2. Выберите пункт **Startup Project** (Запуск проекта) в разделе **Common Properties** (Общие свойства).
3. Выберите пункт **Multiple Startup Projects** (Запускать несколько проектов).
4. Для проектов `DDDesign.Web` и `Sales.Orders.OrderCreated` выберите в поле **Action** (Действие) значение **Start** (Запускать).
5. Переместите проект `DDDesign.Web` в конец списка.
6. Проверьте правильность своих действий, сравнив свои настройки с изображенными на рис. 12.14.
7. Щелкните на кнопке **OK**.

Теперь можно нажать клавишу **F5**, чтобы запустить сеанс отладки приложения. Веб-приложение загрузится в браузер, а ограниченный контекст `Sales` загрузится в консоль. Если теперь внимательно посмотреть на окно консоли, можно увидеть, что `NServiceBus` автоматически выполнил массу грязной работы, создав все необходимое, в том числе и очереди, как показано на рис. 12.15.

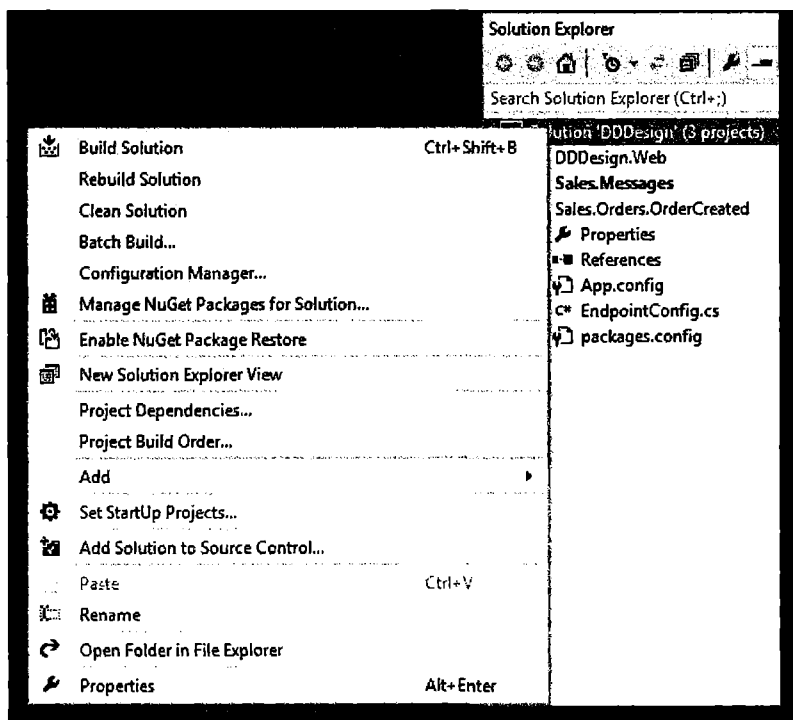


Рис. 12.13. Вызов диалога со свойствами проекта

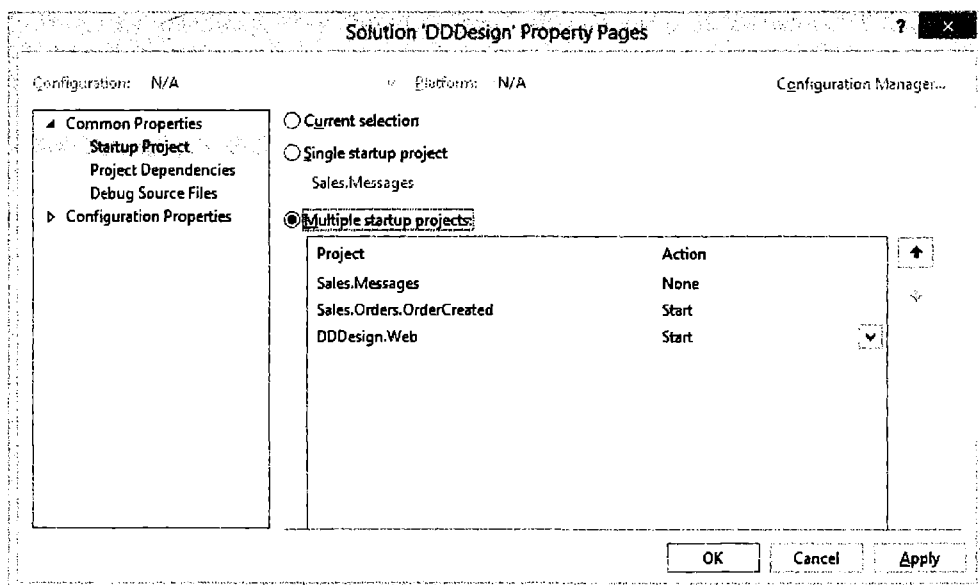


Рис. 12.14. Настройка запуска каждого проекта

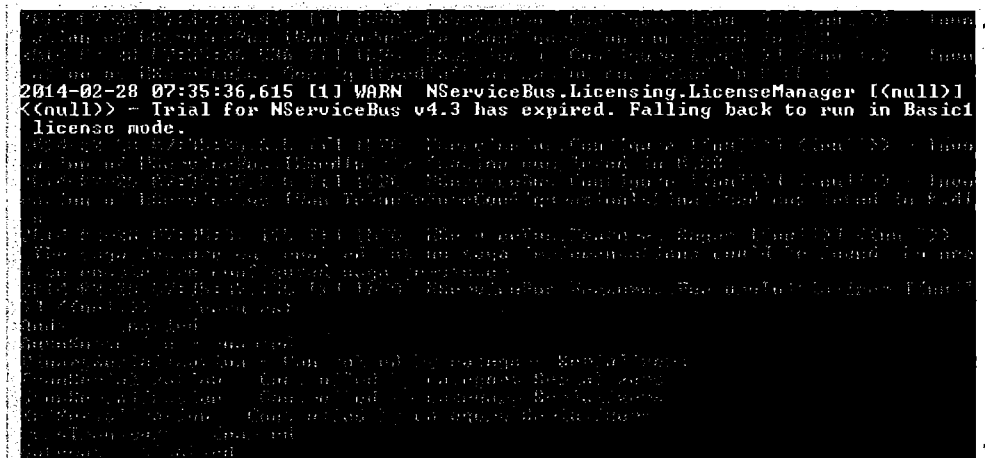


Рис. 12.15. Сервер NServiceBus настроил приложение

Чтобы проверить работу системы, перейдите в браузере по адресу `/orders` и заполните открывшуюся форму. (Вам просто нужно добавить слово «orders» в конец URL в адресной строке браузера.) Затем вернитесь к окну консоли, и вы увидите, что в нем появилась информация о полученном сообщении.

Публикация событий

Теперь, убедившись, что веб-приложение посылает команды, а ограниченный контекст `Sales` благополучно их получает, можно реализовать следующий шаг (под цифрой 2) в диаграмме компонентов, указывающий, что ограниченный контекст `Sales` должен создать заказ и после этого опубликовать событие `OrderCreated`. Для начала следует определить событие `OrderCreated` в проекте `Sales.Messages`. По аналогии с тем, как создавалась команда `PlaceOrder`, просто добавьте папку с именем `Events` в корень проекта и затем добавьте в нее класс с именем `OrderCreated`, как показано на рис. 12.16.

Внутри события `OrderCreated` нужно добавить все необходимые фрагменты информации. Просто добавьте их в класс как свойства, как показано в листинге 12.6.

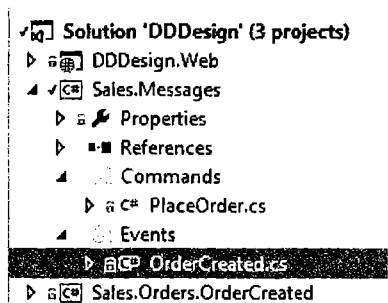


Рис. 12.16. Добавление события `OrderCreated` в проект `Sales.Messages`

Листинг 12.6. Событие OrderCreated

```
using System;
using System.Collections.Generic;

namespace Sales.Messages.Events
{
    public class OrderCreated
    {
        public string OrderId { get; set; }

        public string UserId { get; set; }

        public string[] ProductIds { get; set; }

        public string ShippingTypeId { get; set; }

        public DateTime TimeStamp { get; set; }

        public double Amount { get; set; }
    }
}
```

Теперь, после определения события OrderCreated, его можно опубликовать в любой момент, когда это потребуется. Публикация любых событий выполняется вызовом метода Bus.Publish(), которому передается экземпляр события. В листинге 12.7 представлена измененная версия обработчика PlaceOrderHandler, которая сохраняет заказ в базе данных и затем извещает все заинтересованные стороны о создании заказа, публикуя событие OrderCreated.

Листинг 12.7. Измененная версия обработчика PlaceOrderHandler, публикующая событие OrderCreated

```
using System;
using NServiceBus;
using Sales.Messages.Commands;
using System.Collections.Generic;

namespace Sales.Orders.OrderCreated
{
    public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
    {
        public IBus Bus { get; set; }

        public void Handle(PlaceOrder message)
        {
            var orderId = Database.SaveOrder(
                message.ProductIds, message.UserId, message.ShippingTypeId
            );

            Console.WriteLine(
                @"Created order #{3} : Products:{0} " +
                "with shipping: {1} made by user: {2}",
                String.Join(",", message.ProductIds),
            );
        }
    }
}
```

```

        message.ShippingTypeId, message.UserId, orderId
    );

    var orderCreatedEvent =
        new Sales.Messages.Events.OrderCreated
    {
        OrderId = orderId,
        UserId = message.UserId,
        ProductIds = message.ProductIds,
        ShippingTypeId = message.ShippingTypeId,
        TimeStamp = DateTime.Now,
        Amount = CalculateCostOf(message.ProductIds)
    };

    Bus.Publish(orderCreatedEvent);
}

private double CalculateCostOf(IEnumerable<string> productIds)
{
    // поиск в базе данных и т. п.
    return 1000.00;
}

// База данных может быть любой, причем в разных бизнес-компонентах
// могут использоваться совершенно разные базы данных
public static class Database
{
    private static int Id = 0;

    public static string SaveOrder(IEnumerable<string> productIds,

    string userId, string shippingTypeId)
    {
        var nextOrderId = Id++;
        return nextOrderId.ToString();
    }
}
}

```

Чтобы публикация событий имела смысл, нужно реализовать их обработку и применение правил предметной области.

Подписка на события

Обработка событий, по сути, выполняется так же, как обработка команд. Для демонстрации посмотрим, как реализовать следующий шаг в диаграмме компонентов (см. рис. 12.3) — шаг 3, — вовлекающий подписку ограниченного контекста **Billing** на событие **OrderCreated**, которое публикуется ограниченным контекстом **Sales**. На диаграмме компонентов видно, что ограниченный контекст **Billing** вовлечен также в выполнение шага 4 — публикацию события **PaymentAccepted**.

Поэтому, следуя соглашению об именовании компонентов, следует добавить в решение новый проект библиотеки классов C# с именем `Billing.Payments.PaymentAccepted`. Внутри нового проекта выполните уже знакомые вам шаги:

1. Добавьте ссылку на пакеты `NServiceBus` с помощью диспетчера пакетов (следующая команда должна вводиться в одну строку):

```
Install-Package NServiceBus.Host -ProjectName  
Billing.Payments.PaymentAccepted -Version 4.3.3
```

2. Добавьте соглашения о местонахождении событий в `EndpointConfig`. Этот компонент будет публиковать сообщения, поэтому в том же файле настройте его как издателя, добавив `AsA_Publisher` в цепочку наследования:

```
using NServiceBus;  
namespace Billing.Payments.PaymentAccepted  
{  
    public class EndpointConfig : IConfigureThisEndpoint,  
        AsA_Server, IWantCustomInitialization, AsA_Publisher  
    {  
        public void Init()  
        {  
            Configure.With()  
                .DefiningEventsAs(t => t.Namespace != null  
                    && t.Namespace.Contains("Events"));  
        }  
    }  
}
```

3. Добавьте ссылку в проект `Sales.Message`.
4. Добавьте `OrderCreatedHandler` в корень проекта:

```
using NServiceBus;  
using Sales.Messages.Events;  
using System;  
  
namespace Billing.Payments.PaymentAccepted  
{  
    public class OrderCreatedHandler : IHandleMessages<OrderCreated>  
    {  
        public IBus Bus { get; set; }  
  
        public void Handle(OrderCreated message)  
        {  
            Console.WriteLine(  
                "Received order created event: OrderId: {0}",  
                message.OrderId  
            );  
        }  
    }  
}
```

Есть еще один шаг, связанный с обработкой событий с помощью NServiceBus. Нужно дополнить файл App.config в проекте-подписчике, определив в нем источник событий, на которые он подписывается. В данном случае компонент Billing.Payments.PaymentAccepted подписывается на события OrderCreated, публикуемые компонентом Sales.Orders.OrderCreated. Поэтому файл App.config в проекте Billing.Payments.PaymentAccepted следует дополнить, как показано в листинге 12.8.

Листинг 12.8. Добавление подписки на события NServiceBus в App.config

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
      type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig,
NServiceBus.Core" />
    <section name="UnicastBusConfig"
      type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core" />
    <section name="AuditConfig"
      type="NServiceBus.Config.AuditConfig, NServiceBus.Core" />
  </configSections>
  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Sales.Messages"
        Type="Sales.Messages.OrderCreated"
        Endpoint="Sales.Orders.OrderCreated" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

Увеличение надежности внешних вызовов HTTP с помощью шлюзов сообщений

В главе 11 вы видели, что одной из выгод реактивного решения является улучшенная отказоустойчивость. В качестве примера приводилась ситуация с отключением внешних служб, таких как система платежей. В сценарии на основе RPC попытка оформить заказ потерпела бы неудачу. Но в этом решении, как вы увидите ниже, можно обезопасить себя от отказов внешних служб, используя шлюз сообщений (messaging gateway). Шлюзы сообщений обортывают ненадежные взаимодействия механизмом обмена сообщениями и повторяют попытки обратиться к вышедшим из строя службам, пока они не станут доступными вновь.

Шлюзы сообщений повышают отказоустойчивость

Прежде чем приступить к реализации шлюза сообщений, рассмотрим пример, который поможет понять, зачем вообще нужны шлюзы. Представьте, что класс OrderCreatedHandler реализован, как показано в листинге 12.9.

Листинг 12.9. Реализация, обладающая уязвимостью из-за отсутствия шлюза сообщений

```
public void Handle(OrderCreated message)
{
    Console.WriteLine(
        "Received order created event: OrderId: {0}",
        message.OrderId
    );
    var cardDetails = Database.GetCardDetailsFor(message.UserId);
    var confirmation = PaymentProvider.ChargeCreditCard(
        cardDetails, message.Amount
    );
    if (confirmation.IsSuccess)
    {
        Database.SavePaymentDetails(confirmation);
        Bus.Publish(
            new PaymentAccepted
            {
                OrderId = message.OrderId
            }
        );
    }
    else
    {
        // обработка неудачной попытки получить оплату
    }
}
```

Сможете ли вы найти условия в листинге 12.9, которые с большой долей вероятности вызовут неудовольствие клиентов при попытке оплатить заказ? Если вам нужна подсказка, представьте, как протекают взаимодействия с внешними службами. Что случится, если перед вызовом `Database.SavePaymentDetails()` отключится база данных? Сообщение не будет доставлено и вернется обратно в очередь. Но к этому моменту вы уже списали деньги со счета клиента. При последующих повторных попытках деньги будут списываться со счета клиента снова и снова. Именно по этой причине необходим шлюз сообщений.

По сути, шлюзы сообщений разбивают большие транзакции пополам, то есть если один из вызовов потерпит неудачу, другие операции, уже выполненные к этому моменту (как, например, списание средств со счета клиента), не будут повторяться. В ограниченном контексте `Billing` взаимодействие с платежной системой представляет одну половину транзакции, а запись изменений в базу данных — другую. Для реализации шлюза сообщений нужно всего лишь добавить дополнительное сообщение между ними, что является следующим шагом в данном примере.

Реализация шлюза сообщений

Как упоминалось ранее, шлюз сообщений — это просто шаблон, ситуация, когда один обработчик сообщений разбивается на два или более, если имеют место ненадежные сетевые взаимодействия. Поэтому нет никаких специальных средств,

предоставляемых фреймворками обмена сообщениями, — вам просто нужно создать дополнительные сообщения и предусмотреть их отправку. Тем самым вы гарантируете настолько максимальную изоляцию от рискованных взаимодействий, насколько это возможно.

Начните с определения сообщений

Первой важной задачей `OrderCreatedHandler` является взаимодействие с внешней службой. Вы можете сгенерировать событие, сообщающее, что это произошло. Однако из-за того что шлюз сообщений — это в большей степени особенность реализации, обеспечивающая повышенную отказоустойчивость, а не предметное событие, было бы нежелательно раскрывать эту особенность для других компонентов. Поэтому имеет смысл реализовать сообщение как команду, чтобы гарантировать, что она будет обрабатываться только в одном месте. В данном случае команда называется `RecordPaymentAttempt`. Из диаграммы компонентов вы знаете, что компонент `Billing.Payments.PaymentAccepted` также должен публиковать событие `PaymentAccepted`. Поэтому оба этих события можно добавить в новую библиотеку классов C# с именем `Billing.Messages`. Не забудьте добавить команду в папку `Commands`, а событие — в папку `Events`, как показано на рис. 12.17. Формат этих двух сообщений приводится в листинге 12.10.

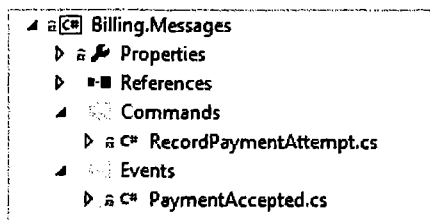


Рис. 12.17. Добавление двух новых сообщений в проект `Billing.Messages`

Листинг 12.10. Сообщения ограниченного контекста `Billing`

```
using System;

namespace Billing.Messages.Events
{
    public class PaymentAccepted
    {
        public string OrderId { get; set; }
    }
}

using System;

namespace Billing.Messages.Commands
{
    public class RecordCompletedPayment
    {
        public string OrderId { get; set; }
```

```

    public PaymentStatus Status { get; set; }
}

public enum PaymentStatus
{
    Accepted,
    Rejected
}
}

```

Затем реализуйте обработку каждого

Теперь все готово к добавлению надежного обработчика `OrderPlacedHandler`, лишенного уязвимостей версии из листинга 12.9. В этой версии вызов службы внешней платежной системы заворачивается в команду `RecordPaymentAttempt`. После сохранения информации об оплате в базе данных наступает черед следующего обработчика. Не забудьте добавить ссылку на `Billing.Messages` в компонент `Billing.Payments.PaymentAccepted`. Затем можно добавить обработчик `OrderCreatedHandler`, представленный в листинге 12.11.

Листинг 12.11. Надежный обработчик `OrderCreatedHandler` со шлюзом сообщений

```

using Billing.Messages.Commands;
using NServiceBus;
using Sales.Messages.Events;
using System;

namespace Billing.Payments.PaymentAccepted
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated>
    {
        // зависимость, внедряемая фреймворком NServiceBus
        public IBus Bus { get; set; }

        public void Handle(OrderCreated message)
        {
            Console.WriteLine(
                "Received order created event: OrderId: {0}",
                message.OrderId
            );

            var cardDetails = Database.GetCardDetailsFor(
                message.UserId
            );

            var conf = PaymentProvider.ChargeCreditCard(
                cardDetails, message.Amount
            );

            var command = new RecordPaymentAttempt
            {
                OrderId = message.OrderId,

```

```

        Status = conf.Status
    };

    Bus.SendLocal(command);
}
}

// исключительно демонстрационный класс
// остерегайтесь статических зависимостей
public static class PaymentProvider
{
    public static PaymentConfirmation ChargeCreditCard(
        CardDetails details, double amount)
    {
        // чтобы сохранить фокус примера на том,
        // что относится к обсуждаемой теме
        return new PaymentConfirmation
        {
            Status = PaymentStatus.Accepted
        };
    }
}

public class PaymentConfirmation
{
    public PaymentStatus Status { get; set; }
}

public static class Database
{
    public static CardDetails GetCardDetailsFor(string userId)
    {
        // чтобы сохранить фокус примера на том,
        // что относится к обсуждаемой теме
        return new CardDetails();
    }
}

public class CardDetails
{
    // ...
}
}

```

Здесь следует отметить одну маленькую деталь — вызов `Bus.SendLocal()`. В этом есть определенный смысл, потому что посылающий компонент также должен обработать команду. Метод `SendLocal()` просто посылает сообщение в тот же узел `NServiceBus`, который прислал сообщение, где оно будет передано соответствующему обработчику.

Чтобы скомпилировать решение на этом этапе, добавьте обработчик команды `RecordPaymentAttempt`, имитирующий запись информации о платеже в базе данных. Этот обработчик представлен в листинге 12.12. Его можно поместить в один

файл с классом `OrderCreatedHandler`, если это поможет вам мысленно связать поток сообщений.

Листинг 12.12. Обработчик `RecordPaymentHandler`, находящийся по ту сторону шлюза сообщений

```
using System;
using Billing.Messages.Commands;
using NServiceBus;

namespace Billing.Payments.PaymentAccepted
{
    public class RecordPaymentAttemptHandler :
        IHandleMessages<RecordPaymentAttempt>
    {
        // зависимость, внедряемая фреймворком NServiceBus
        public IBus Bus { get; set; }

        public void Handle(RecordPaymentAttempt message)
        {
            Database.SavePaymentAttempt(
                message.OrderId, message.Status
            );

            if (message.Status == PaymentStatus.Accepted)
            {
                var evnt = new Billing.Messages.events.PaymentAccepted
                {
                    OrderId = message.OrderId
                };
                Bus.Publish(evnt);
                Console.WriteLine(
                    "Received payment accepted notification for Order:" +
                    " {0}. Published PaymentAccepted event",
                    message.OrderId
                );
            }
            else
            {
                // опубликовать событие отвергнутой попытки платежа
            }
        }

        public static class Database
        {
            public static void SavePaymentAttempt(string orderId,
                PaymentStatus status)
            {
                // .. сохранить в выбранной базе данных
            }
        }
    }
}
```

Управление повторными попытками отправки сообщений

Теперь у вас есть отличная возможность в полной мере осознать ценность сообщений. Далее вы увидите, как в действительности шина повторяет отправку сообщений, когда сталкивается с недоступностью внешней службы. При этом данный пример будет полностью использовать возможности только что реализованного шлюза сообщений. Чтобы симитировать отказ, вам нужно изменить фиктивную реализацию службы внешней платежной системы так, чтобы она возбуждала исключение в первых двух попытках обратиться к ней, а затем благополучно принимала третий запрос.

Имитация отказа

Если вы измените реализацию `PaymentProvider`, как показано в листинге 12.13, вы будете практически готовы убедиться в отказоустойчивости своей реактивной системы.

Листинг 12.13. Версия `PaymentProvider`, имитирующая отказы внешней службы

```
public static class PaymentProvider
{
    private static int Attempts = 0;

    public static PaymentConfirmation ChargeCreditCard(
        CardDetails details, double amount)
    {
        if (Attempts < 2)
        {
            Attempts++;
            throw new Exception(
                "Service unavailable. Down for maintenance."
            );
        }
        return new PaymentConfirmation
        {
            Status = PaymentStatus.Accepted
        };
    }
}
```

Перед тестированием первого обработчика, выше в этой главе, вы должны были настроить запуск каждого компонента `NServiceBus` вместе с решением. То же самое нужно проделать и теперь в отношении компонента `Billing.Payment.PaymentAccepted`, чтобы можно было протестировать работу шлюза сообщений. Измените настройки запуска, как показано на рис. 12.18.

Теперь можно нажать клавишу `F5` и запустить приложение. После этого перейдите в браузере на страницу `/orders` и разместите заказ. Особое внимание обратите на окно консоли для компонента `Billing.Payments.PaymentAccepted`. Сначала вы увидите, что `OrderCreatedHandler` принял событие `OrderCreated`. Потом — что событие `PaymentAccepted` было опубликовано, как показано на рис. 12.19. Но если

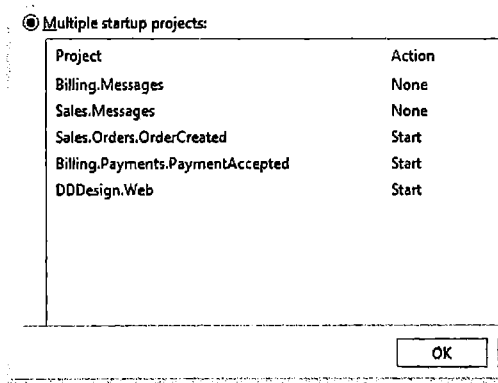


Рис. 12.18. Настройка запуска всех серверов NServiceBus

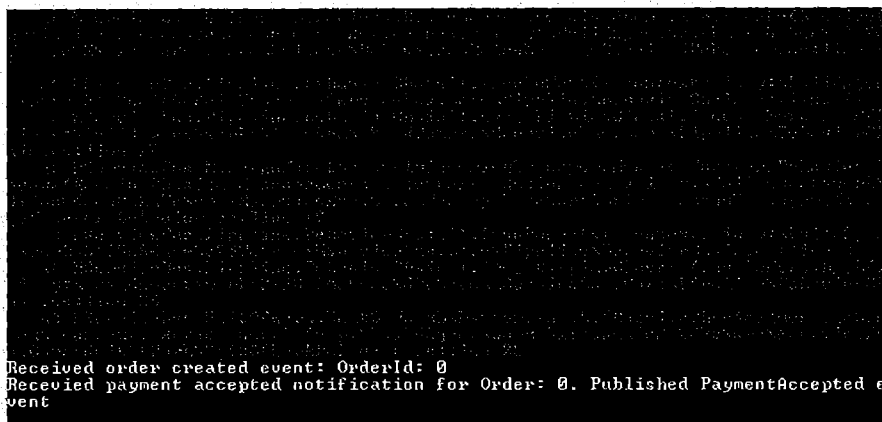


Рис. 12.19. Все выглядит так, как если бы попытка оплатить заказ увенчалась успехом

прокрутить окно консоли вверх, вы увидите, что в действительности было произведено три попытки. На рис. 12.20 показаны сообщения, которые вывел фреймворк NServiceBus в консоль, когда не смог обработать первое сообщение. Обратите внимание, что он даже вывел тип исключения и текст сообщения в нем, которое возбудил фиктивный класс, имитирующий работу службы платежной системы: `System.Exception: Service unavailable. Down for maintenance.`

Повторные попытки второго уровня

Если платежная система отключилась на длительное время, например на десять минут, фреймворк NServiceBus может достигнуть предельного числа повторных попыток отправки сообщения. К счастью, NServiceBus включает механизм, который называется *повторные попытки второго уровня* (2nd level retries), позволяющий определить более длительные периоды между повторными попытками. В данном примере было бы полезно продолжить повторять попытки отправки

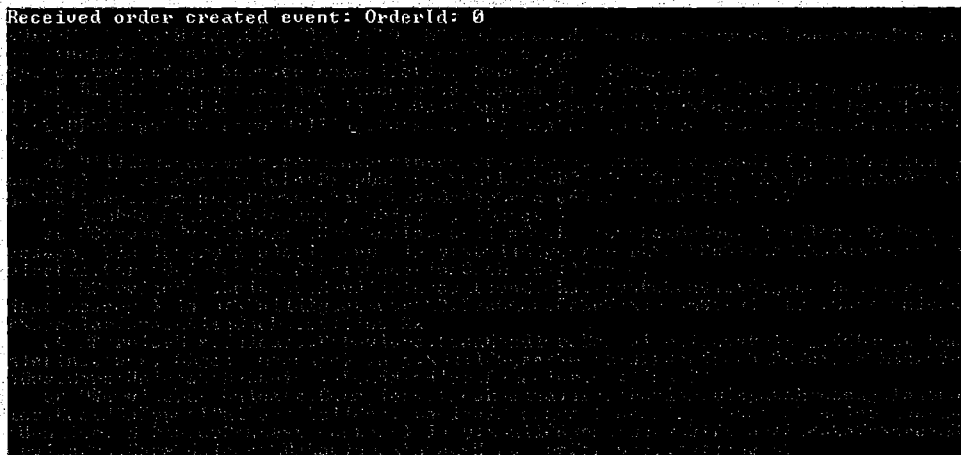


Рис. 12.20. В действительности две первые попытки потерпели неудачу

сообщения каждые десять минут в течение часа. Соответствующие настройки приводятся в листинге 12.14.

Листинг 12.14. Настройка повторных попыток второго уровня в NServiceBus

```
<configSections>
  <section name="SecondLevelRetriesConfig"
    type="NServiceBus.Config.SecondLevelRetriesConfig, NServiceBus.Core"/>
</configSections>

<SecondLevelRetriesConfig Enabled="true"
  TimeIncrease ="00:10:00"
  NumberOfRetries="6" />
```

Иногда успешная обработка сообщений в принципе невозможна, потому что система не может обработать их. Это происходит, если, например, сообщение содержит ошибочные данные или из базы данных была удалена информация, необходимая для обработки сообщения. Такие сообщения называют *подозрительными* (poison messages), и они неизбежно оказываются в очереди *error*. Подробнее об очереди *error*, а также о ее мониторинге и обработке сообщений в этой очереди будет рассказываться далее в этой главе.

Потенциальная непротиворечивость на практике

В данной точке в процессе размещения заказа система может находиться в противоречивом, несогласованном состоянии. Даже при том, что заказ создан ограниченным контекстом *Sales*, а оплата обработана ограниченным контекстом *Billing*, ограниченный контекст *Shipping* еще ничего не знает о заказе и не организовал его доставку. В некоторых системах, где вся последовательность операций выпол-

няется в одной атомарной транзакции и либо все операции завершаются успехом, либо все терпят неудачу, такое положение вещей просто невозможно. То есть данный случай можно рассматривать как пример потенциальной непротиворечивости. Это подходящий момент узнать, как действовать в ситуации потенциальной непротиворечивости, когда интеграция ограниченных контекстов осуществляется посредством сообщений.

Обработка несогласованного состояния

Потенциальная непротиворечивость может привести к нежелательным последствиям. Например, если платежная система отвергла попытку оплатить заказ, нельзя будет просто откатить транзакцию и отменить создание заказа (как в обычных, однозначно согласованных системах); заказ уже был создан в ходе предыдущей транзакции, в другом компоненте и в настоящее время хранится в базе данных того компонента. Что можно сделать в данном случае, так это двинуться вперед, к новому состоянию. Вы могли бы сообщить клиенту, что заказ не может быть выполнен, потому что провалилась попытка получить оплату. В идеале можно было бы сообщить об этом сразу, пока клиент еще пытается оформить заказ. Но не забывайте, что вы пытаетесь создать масштабируемое, отказоустойчивое решение и потому должны чем-то жертвовать. Недовольство нескольких клиентов, потерпевших неудачу при попытке разместить заказ, в обмен на положительные впечатления у всех остальных — часто вполне приемлемый компромисс.

Когда система оказывается в несогласованном состоянии, нужно двигаться дальше, к новому состоянию, отражающему пожелания бизнеса или фактически протекающие процессы в предметной области, которые моделирует система.

Движение вперед, к новым состояниям

Чтобы решить проблему невозможности оформить заказ в состоянии потенциальной непротиворечивости, можно вместе со специалистами предприятия создать новую диаграмму компонентов, учитывающую сценарий, когда невозможно получить оплату, чтобы понять, к какому состоянию следует двигаться. Представим, что мы действительно поговорили со специалистами и нам сказали, что если платежная система отвергла попытку списать деньги со счета клиента, работники отдела продаж (Sales) должны информировать клиента, что заказ аннулирован. На рис. 12.21 изображена часть диаграммы компонентов, соответствующая данному сценарию.

Как показано на рис. 12.21, для решения проблемы потенциальной непротиворечивости в этом сценарии достаточно просто опубликовать событие `PaymentRejected` в ограниченном контексте `Billing` и обработать его в ограниченном контексте `Sales`, отправив электронное письмо клиенту. Чтобы реализовать этот сценарий использования, нужно всего лишь отправить и обработать несколько сообщений. Чтобы избежать повторений, не показанных здесь, вы можете реализовать все это самостоятельно, используя предыдущие примеры как образец.

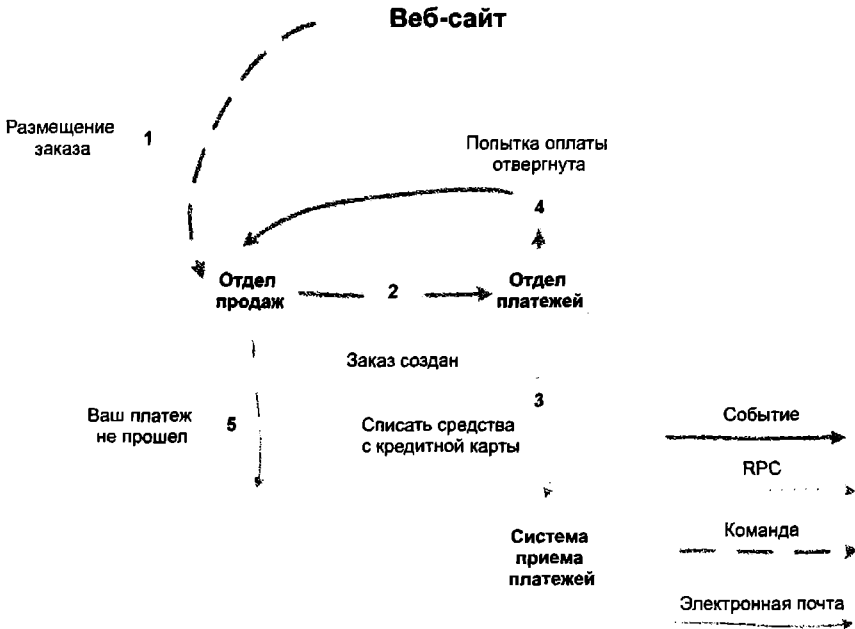


Рис. 12.21. Сценарий, когда платежная система отвергает попытку списать деньги со счета клиента

ПРИМЕЧАНИЕ

Если вы все еще не уверены, стоит ли отказываться от транзакций в пользу потенциальной непротиворечивости, прочитайте статью Джимми Богарда (Jimmy Bogard), опубликованную им в его блоге в 2013 году, где он пытается развеять распространенные страхи (<http://lostechies.com/jimmybogard/2013/05/09/ditching-two-phased-commits/>). Вам определенно стоит потратить пять минут, чтобы прочитать ее и осознать упомянутые в ней идеи.

Ограниченные контексты хранят все необходимые им данные локально

Следующие шаги в диаграмме компонентов (с номерами 4, 5 и 6) подключают ограниченный контекст **Shipping**, организующий доставку заказа. Но здесь есть одна проблема. Взгляните еще раз на событие **PaymentAccepted**:

```
public class PaymentAccepted
{
    public string OrderId { get; set; }
}
```

Как организовать доставку, имея только идентификационный номер заказа? К сожалению, никак, поэтому нужно еще раз внимательно рассмотреть имеющиеся

возможности. Для простоты представим, что каждый пользователь имеет единственный адрес, и этот адрес хранится в базе данных бизнес-компонента `Sales.Customers`. Одно из решений состоит в том, чтобы выполнить RPC-запрос к бизнес-компоненту `Sales.Customers` по протоколу HTTP и получить необходимые данные. Но тогда возникает временная зависимость. Если база данных или конечная точка RPC бизнес-компонента `Sales.Customers` отключится, ограниченный контекст `Shipping` не сможет выполнить свою работу. Если другие ограниченные контексты будут напрямую обращаться к базе данных `Sales.Customers`, разработчики ограниченного контекста `Sales` не смогут реструктурировать или изменять базу данных в соответствии со своими потребностями, не беспокоясь об отрицательном влиянии на другие ограниченные контексты.

Другая возможность — передавать адрес клиента в событии `PaymentAccepted`. Это решение также опасно, потому что фактически объединяет два сообщения, добавляя свойства из первого сообщения во второе. В сценарии, когда посылаются четыре сообщения, четвертое сообщение должно содержать все свойства из первых трех. Это может привести к образованию тесной зависимости и сложностям во время отладки, когда понадобится выяснить, откуда пришли данные. Как прагматик, вы могли бы попробовать использовать это решение в нескольких местах, но это крайне нежелательно, если только ваше решение не продиктовано какими-то специфическими случаями использования.

Хранение стоит недорого — храните локальную копию

Чтобы избежать образования зависимостей в этой ситуации, можно реализовать хранение всех данных каждого ограниченного контекста локально. Часто это неплохой компромисс, потому что в наши дни хранение стоит недорого. Например, на момент написания этих строк жесткий диск емкостью 1 Тбайт стоил около \$50. Это означает, что оба ограниченных контекста, `Sales` и `Shipping`, должны хранить адрес клиента. Чуть ниже вы узнаете, какие проблемы влечет это решение, а пока просто постарайтесь сосредоточиться на реализации ограниченного контекста `Shipping` с использованием этого подхода. А решением проблем мы займемся позже.

На рис. 12.22 изображен процесс регистрации пользователя. Обратите внимание, что здесь публикуется событие `NewBusinessUserRegistered`, которое обрабатывается двумя ограниченными контекстами, подписавшимися на него. Бизнес-компонент `BusinessCustomers` в ограниченном контексте `Shipping` получает идентификатор клиента и адрес. Бизнес-компонент `BusinessCampaigns` в ограниченном контексте `Marketing` получает идентификатор клиента, его адрес и область, в которой работает предприятие. Оба ограниченных контекста хранят данные локально, несмотря на то что одни и те же данные хранятся в нескольких местах.

ПРИМЕЧАНИЕ

Для реализации ограниченного контекста `Shipping` требуется выполнить те же действия, что выполнялись при реализации ограниченного контекста `Billing`, поэтому некоторые детали были опущены. Если вам понадобится освежить память, вернитесь к предыдущим разделам. Также не забывайте, что если вам понадобится помощь, вы всегда сможете задать свой вопрос на дискуссионном форуме на сайте издательства Wrox.

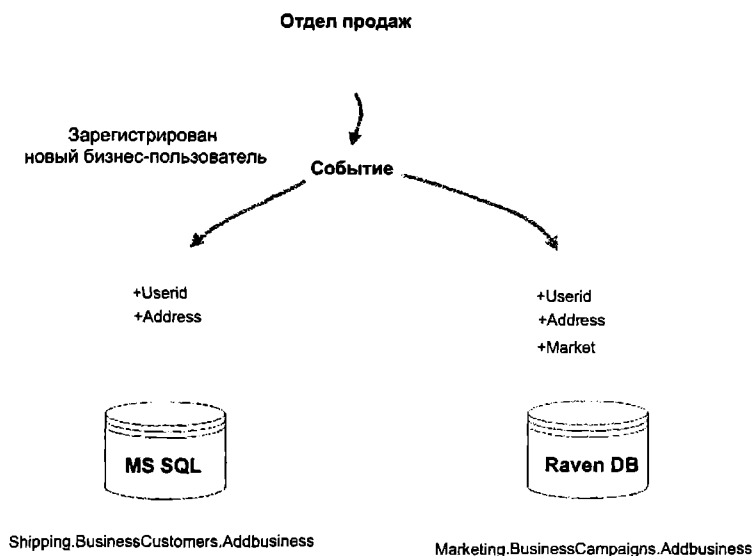


Рис. 12.22. Несколько ограниченных контекстов хранят одни и те же данные локально

В реализации оставшейся части системы электронной коммерции, создание которой было начато в этой главе, будет использоваться локальное хранилище. Но прежде нужно создать компонент `Shipping.BusinessCustomers.ArrangeShipping`, подписывающийся на событие `OrderCreated`, издаваемое ограниченным контекстом `Sales`, и на событие `PaymentAccepted`, издаваемое ограниченным контекстом `Billing`. Для этого выполните следующие шаги.

1. Создайте новую библиотеку классов C# с именем `Shipping.BusinessCustomers.ShippingArranged`.
2. Добавьте ссылки на проекты `Sales.Messages` и `Billing.Messages`.
3. Добавьте зависимости `NServiceBus` с помощью диспетчера пакетов NuGet.
4. Скопируйте соглашения по размещению команд и событий в `EndpointConfig`. Также добавьте в определение `EndpointConfig` наследование класса `Asa_Publisher` и интерфейса `IWantCustomInitialization`, как показано в листинге 12.15.
5. Добавьте в проект классы `OrderCreatedHandler` и `PaymentAcceptedHandler`, представленные в листинге 12.16.
6. Добавьте настройки подписки для событий `OrderCreated` и `PaymentAccepted` в файл `App.config` проекта. После этого содержимое вашего файла `App.config` должно напоминать листинг 12.17.
7. Настройте запуск проекта.

8. Добавьте библиотеку классов C# `Shipping.Messages` с папкой `Events`, содержащей событие `ShippingArranged`, представленное в листинге 12.18.
9. Добавьте ссылку на проект `Shipping.Messages` в компонент `Shipping.BusinessCustomers.ArrangeShipping`.
10. Настройте в Visual Studio запуск проекта `Shipping.BusinessCustomers.ArrangeShipping`.

Листинг 12.15. `EndpointConfig` для компонента `ShippingArranged`

```
using NServiceBus;

namespace Shipping.BusinessCustomers.ShippingArranged
{
    public class EndpointConfig : IConfigureThisEndpoint,
        AsA_Server, AsA_Publisher, IWantCustomInitialization
    {
        public void Init()
        {
            Configure.With()
                .DefiningCommandsAs(t => t.Namespace != null
                    && t.Namespace.Contains("Commands"))
                .DefiningEventsAs(t => t.Namespace != null
                    && t.Namespace.Contains("Events"));
        }
    }
}
```

Листинг 12.16. `OrderCreatedHandler` и `PaymentAcceptedHandler`

```
using Billing.Messages.Events;
using NServiceBus;
using Sales.Messages.Events;
using System;
using System.Linq;
using System.Collections.Generic;

namespace Shipping.BusinessCustomers.ShippingArranged
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated>
    {
        // зависимость, внедряемая фреймворком NServiceBus
        public IBus Bus { get; set; }

        public void Handle(OrderCreated message)
        {
            Console.WriteLine(
                "Shipping BC storing: Order: {0} User: {1} " +
                "Shipping Type: {2}",
                message.OrderId, message.UserId, message.ShippingTypeId
            );
        }
    }
}
```

```

        var order = new ShippingOrder
        {
            UserId = message.UserId,
            OrderId = message.OrderId,
            ShippingTypeId = message.ShippingTypeId
        };
        ShippingDatabase.AddOrderDetails(order);
    }
}

public class PaymentAcceptedHandler : IHandleMessages<PaymentAccepted>
{
    // зависимость, внедряемая фреймворком NServiceBus
    public IBus Bus { get; set; }

    public void Handle(PaymentAccepted message)
    {
        var address = ShippingDatabase.GetCustomerAddress(
            message.OrderId
        );

        var confirmation = ShippingProvider.ArrangeShippingFor(
            address, message.OrderId
        );

        if (confirmation.Status == ShippingStatus.Success)
        {
            var evnt = new Shipping.Messages.events.ShippingArranged
            {
                OrderId = message.OrderId
            };
            Bus.Publish(evnt);
            Console.WriteLine(
                "Shipping BC arranged shipping for Order:" +
                " {0} to: {1}",
                message.OrderId, address
            );
        }
        else
        {
            // ..
        }
    }
}

public static class ShippingDatabase
{
    private static List<ShippingOrder> Orders =
        new List<ShippingOrder>();

    public static void AddOrderDetails(ShippingOrder order)

```

```
{
    Orders.Add(order);
}

public static string GetCustomerAddress(string orderId)
{
    /*
     * Реализуйте здесь сохранение адреса пользователя, когда
     * ограниченный контекст Sales публикует соответствующее
     * событие (событие добавьте самостоятельно)
     */
    var order = Orders.Single(o => o.OrderId == orderId);
    return string.Format(
        "{0}, 55 DDEsign Street",
        Order.UserId
    );
}
}

public class ShippingOrder
{
    public string UserId { get; set; }

    public string OrderId { get; set; }

    public string ShippingTypeId { get; set; }
}

public static class ShippingProvider
{
    public static ShippingConfirmation ArrangeShippingFor(
        string address, string referenceCode)
    {
        return new ShippingConfirmation
        {
            Status = ShippingStatus.Success
        };
    }
}

public class ShippingConfirmation
{
    public ShippingStatus Status { get; set; }
}

public enum ShippingStatus
{
    Success,
    Failure
}
}
```

Листинг 12.17. App.config для компонента ShippingArranged

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig,
NServiceBus.Core" />
    <section name="UnicastBusConfig"
type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core" />
    <section name="AuditConfig"
type="NServiceBus.Config.AuditConfig,
NServiceBus.Core" />
  </configSections>
  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Sales.Messages"
Type="Sales.Messages.OrderCreated "
Endpoint="Sales.Orders.OrderCreated" />
      <add Messages="Billing.Messages"
Type="Billing.Messages.PaymentAccepted"
Endpoint="Billing.Payments.PaymentAccepted" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

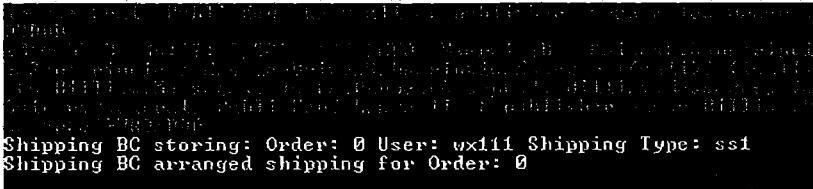
Листинг 12.18. Событие ShippingArranged

```
using System;

namespace Shipping.Messages
{
    public class ShippingArranged
    {
        public string OrderId { get; set; }
        /*
         * Добавьте сюда другие поля, такие как дата/диапазон дат,
         * которые могут понадобиться, в зависимости от API
         * службы транспортной компании, с которой вы сотрудничаете
         */
    }
}
```

Если вы все сделали правильно, вся ваша цепочка теперь должна работать корректно вплоть до момента организации доставки. Запустите приложение нажатием клавиши F5, перейдите на страницу /orders и заполните форму. После этого должны появиться три окна консоли — по одному для каждого компонента. В окне консоли Shipping.BusinessCustomer.ArrangeShipping вы должны увидеть, что компонент ArrangeShipping сохранил значения UserId и OrderId, полученные

от ограниченного контекста `Sales` в сообщении `OrderCreated`, а затем использовал значение `UserId` для поиска адреса, когда обрабатывал событие `PaymentAccepted`. Вывод в консоли должен выглядеть, как показано на рис. 12.23 (со значениями, совпадающими с теми, что были введены в форме).



```
Shipping BC storing: Order: 0 User: wx111 Shipping Type: ssl
Shipping BC arranged shipping for Order: 0
```

Рис. 12.23. Ограниченный контекст `Shipping` сохраняет данные локально и использует их для организации доставки

Проблемы дублирования общих данных

Теперь, когда вы знаете, как сохранить данные локально, в каждом бизнес-компоненте, можно поговорить о проблемах, на которые многие жалуются, когда сталкиваются с таким существенным изменением философии. Множество людей считают проблемой дублирование данных, потому что их волнует возможная неэффективность такого решения. Также встает вопрос обновления данных, таких как адреса клиентов, для которых имеется множество дублирующих копий. Системы обмена сообщениями, в частности, поднимают проблему доставки сообщений с нарушением их порядка. Представьте, что пользователь зарегистрировался и тут же попытался разместить заказ. Что произойдет, если ограниченный контекст `Shipping` получит предложение организовать доставку до того, как сохранит адрес пользователя?

Не забывайте, что решение хранить данные локально не является идеальным. Но оно часто ослабляет зависимости и служит отличной основой для улучшения масштабируемости, отказоустойчивости и увеличения скорости разработки. Наличие централизованных точек управления логикой или данными в распределенных системах по этим причинам может быть нежелательным. Однако возникающие проблемы достаточно серьезны.

Данные концептуально различны

Два бизнес-компонента могут хранить цену товара. Например, ограниченный контекст `Billing` хранит последнюю цену, чтобы при размещении заказов пользователи платили текущую цену. Но отдел продаж (ограниченный контекст `Sales`) наверняка пожелает сохранить цену продукта после размещения заказа. Представьте себе покупку в магазине. Когда вы возвращаете товар и требуете вернуть деньги, вы предъявляете чек и получаете уплаченные деньги, а не последнюю цену товара. Это пример, почему в разных контекстах данные могут оказаться концептуально разными даже при том, что выглядят одинаковыми, — они используются в разных целях и изменяются по разным причинам.

Избегайте несогласованности, используя сроки действия

Предприятия часто изменяют цены на товары для их продвижения, при снижении популярности или с появлением на рынке новых альтернатив. В таких случаях цена в одном ограниченном контексте может измениться раньше, чем в другом. Проблема в том, что пользователи могут видеть на веб-сайте одну цену, а платить другую после добавления товара в корзину. Эту проблему легко исправить, добавив в сообщение срок действия. Ниже демонстрируется событие `PriceUpdated` с полями `AvailableFrom` и `AvailableTo`, определяющими период действия этого события:

```
public class PriceUpdated
{
    public string ProductId { get; set; }

    public double Price { get; set; }

    public DateTime AvailableFrom { get; set; }

    public DateTime AvailableTo { get; set; }
}
```

Когда сообщение имеет ограниченный срок действия, оно должно посылаться как можно скорее. Чем раньше будет послано извещение каждому ограниченному контексту, тем больше времени будет для подготовки к вступлению сообщения в силу. Часто есть возможность согласовать свою стратегию с бизнес-правилами. В данном примере можно было бы поговорить с заинтересованными лицами и узнать, как много времени они отводят для подготовки к изменению цен. После этого можно было бы условиться с ними, что вы готовы гарантировать своевременное обновление всех цен, если получите уведомление, например, не позднее чем за 24 часа.

Экономика дублирования данных

Вы только что узнали, что данные могут быть концептуально разными, поэтому в некоторых случаях неправильно употреблять слово *дублирование*. Кроме того, необходимо учитывать некоторые факторы, определяющие экономическую эффективность такого дублирования. Как упоминалось выше, жесткий диск емкостью 1 Тбайт стоит около \$50. Чем больше объем закупаемой вами партии, тем выгоднее становится приобретение. Услуги облачных хранилищ также имеют вероятно низкую цену.

Однако то, за что вы не платите явно, имеет свою скрытую цену. Сколько вы теряете при наличии программных компонентов и групп разработчиков, тесно связанных между собой из-за использования одних и тех же данных? Подумайте о необходимости масштабирования; попробуйте представить, насколько быстрее будет создана система независимыми группами разработчиков. Подобные затраты трудно подсчитать, но, учитывая дешевизну хранения данных, разработчики обычно рады заплатить за дублирование данных, чтобы получить возможность сосредоточиться на более быстром создании ценностей для предприятия.

Нарушение порядка доставки сообщений

Что может случиться, если пользователь зарегистрировался, тут же разместил заказ и ограниченный контекст `Shipping` получил сообщение `PaymentAccepted` до того, как к нему поступило сообщение с адресом пользователя? В подобных ситуациях можно просто вернуть сообщение в очередь и попробовать обработать его позднее, когда будет доставлено и обработано сообщение с адресом. Часто именно так решается проблема нарушения порядка в поступлении сообщений.

Объединяем все вместе в пользовательском интерфейсе

Передача данных в систему обмена сообщениями, о чем до настоящего момента рассказывалось в этой главе, является лишь частью истории. Необходимо также иметь возможность извлекать данные из этой системы. Для этого требуется совершенно иной образ мышления в сравнении с некоторыми традиционными подходами, особенно когда данные хранятся в каждом бизнес-компоненте локально. На диаграмме контейнеров (см. рис. 12.4) есть подсказка, как это сделать: ограниченные контексты часто должны экспортировать свои данные по протоколу `HTTP`, чтобы обеспечить их доступность для веб-приложений.

В следующей главе вы познакомитесь с множеством эффективных приемов и конкретными примерами создания `HTTP API` для экспортирования функциональных возможностей системы и данных. А в этом разделе вы узнаете о некоторых понятиях, достоинствах и недостатках, особенно важных в системах обмена сообщениями. Однако многие другие идеи, представленные в следующей главе, до определенной степени сохранят свою актуальность.

ПРИМЕЧАНИЕ

Подробнее о создании пользовательских интерфейсов поверх `HTTP API` и систем обмена сообщениями рассказывается в главе 23 «Конструирование пользовательских интерфейсов приложения». В ней приводится множество практических примеров.

Бизнес-компоненты должны иметь свои прикладные интерфейсы

Если вы действуете по принципу «каждый бизнес-компонент должен иметь свою частную базу данных», единственный разумный способ экспортировать данные для нужд веб-приложения — реализовать в каждом бизнес-компоненте свой собственный прикладной программный интерфейс (API). На рис. 12.24 изображена эта предлагаемая схема.

Как показано на рис. 12.24, в некоторых бизнес-компонентах решено экспортировать данные двум веб-приложениям. Некоторые другие бизнес-компоненты, напротив, не предоставляют API, потому что не экспортируют ничего; они просто получают данные и выполняют другие задачи. Внутри своих бизнес-компонентов вы можете поступать как вам заблагорассудится. Число проектов, число API и даже число используемых технологий зависит только от ваших суждений и потребностей проекта.

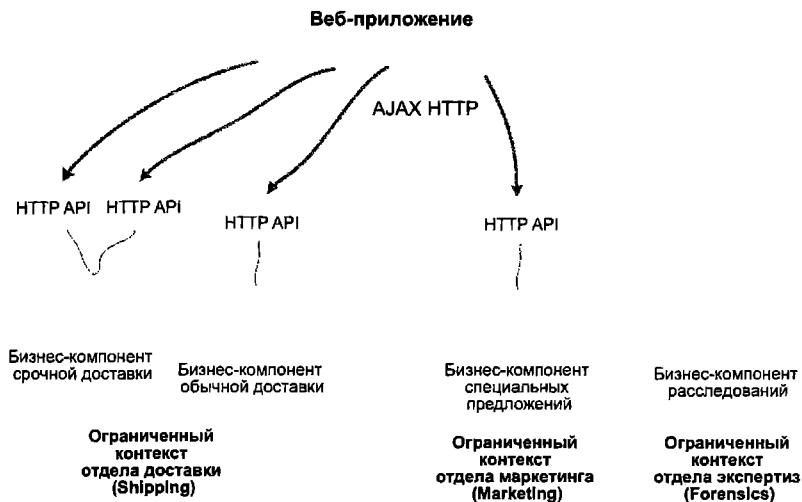


Рис. 12.24. Бизнес-компоненты имеют собственные прикладные интерфейсы

Будьте осторожны, организуя взаимодействия на стороне сервера

Может показаться заманчивым организовать обращение ко всем API на стороне сервера (внутри контроллера) и возвращать клиенту веб-страницу с объединенной моделью. К сожалению, это может свести на нет все усилия, направленные на устранение тесных связей между компонентами и улучшение отказоустойчивости системы, потому что при таком решении появляется еще одна точка, где образуется тесная связь. Во-первых, если один из множества API окажется недоступен, это может привести к ошибке и страница не будет передана клиенту. Задумайтесь, действительно ли это так необходимо? Представьте, что вы конструируете страницу, отображающую каталог товаров, и в нижней ее части перечисляются специальные предложения. На рис. 12.25 показан пример, как разные части страницы получают данные из разных API.

Как вы думаете, хотели бы владельцы предприятия, чтобы клиенты видели на экране сообщение об ошибке, мешающее им потратить деньги, если в ограниченном контексте Marketing наблюдаются проблемы, не позволяющие ему вернуть информацию о специальных предложениях? Многие из них наверняка предпочли бы, чтобы приложение отображало основной список товаров, даже при невозможности показать специальные предложения. Да, такое можно реализовать, управляя взаимодействиями на стороне сервера, но для этого потребуются приложить дополнительные усилия и не факт, что ничего не будет упущено. Кроме того, контроллер в веб-приложении, управляющий взаимодействиями, — это дополнительный компонент, который сам может выйти из строя. По этим причинам некоторые разработчики предпочитают загружать данные непосредственно в страницу, используя технологию AJAX.

Наполнение пользовательского интерфейса данными с помощью AJAX

Взгляните еще раз на рис. 12.25, здесь видно, что каждый раздел страницы извлекает соответствующие фрагменты данных, используя разные API. Почему бы просто не выполнять веб-запросы AJAX непосредственно из страницы и не избавиться от необходимости организовывать взаимодействия на стороне сервера? Прикладные интерфейсы могли бы возвращать данные в легковесном формате JSON, а веб-страницы — использовать ваши любимые библиотеки JavaScript для управления данными и их отображением. Такой способ наполнения пользовательского интерфейса особенно хорошо подходит, когда требуется быстро создать одностраничное приложение или имеются универсальные API, используемые многими другими веб-сайтами.

Наполнение пользовательского интерфейса с помощью AJAX HTML

Прикладные интерфейсы могли бы также возвращать данные не в формате JSON или XML, а сразу в виде разметки HTML, пригодной для непосредственного

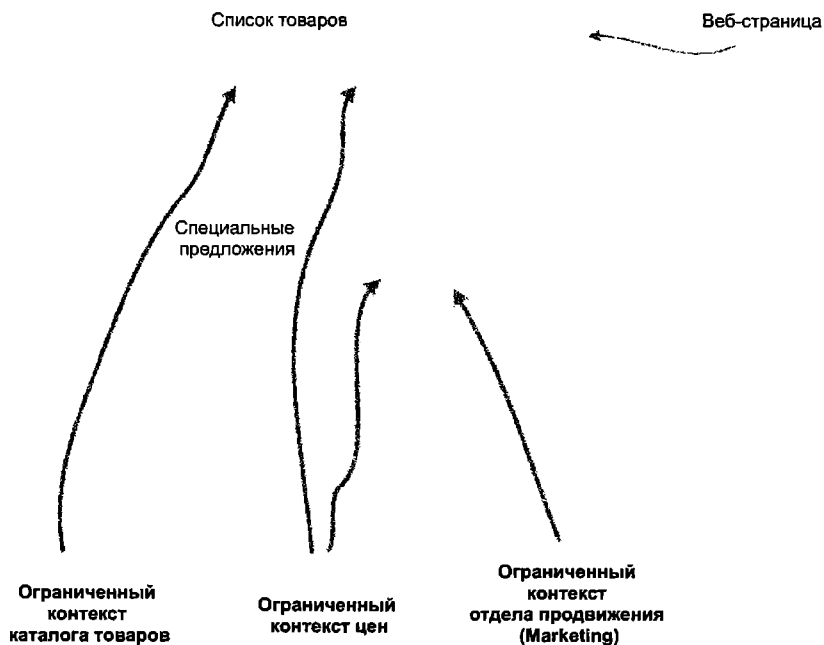


Рис. 12.25. Страницы получают данные из разных API

добавления в страницу. Сторонники такого способа наполнения пользовательского интерфейса предпочитают переложить на каждый ограниченный контекст решение о том, как должен выглядеть тот или иной раздел страницы. Взгляните еще раз на рис. 12.25: если ограниченный контекст **Marketing** будет возвращать разметку HTML, он сможет управлять внешним видом раздела страницы со специальными предложениями и даже изменять оформление этого раздела по мере необходимости, возвращая измененную разметку HTML.

Несмотря на то что прием наполнения пользовательского интерфейса разметкой HTML дает каждому ограниченному контексту дополнительные рычаги управления, может оказаться очень сложно обеспечить единство стиля для всей страницы, что легко достигается, когда вся разметка HTML сосредоточена в одном файле. Кроме того, такой подход может вызывать сложности, когда прикладным интерфейсом пользуется несколько веб-приложений. Но как бы то ни было, такой вариант нередко выбирается некоторыми группами разработчиков, и вы определенно должны подумать о преимуществах его применения в своих проектах.

Экспортирование API для использования внешними приложениями

Прикладные интерфейсы (API) не всегда предназначены только для внутреннего использования. Иногда к ним обращаются разные приложения, созданные другими разработчиками. В некоторых компаниях прикладной интерфейс вообще является главным продуктом. В следующей главе вы узнаете, как создавать прикладные интерфейсы HTTP, используя такие шаблоны проектирования, как REST, чтобы получить возможность экспортировать их для внешних потребителей.

Сопровождение приложений, использующих обмен сообщениями

После создания системы обмена сообщениями появляются другие серьезные проблемы, с которыми придется иметь дело. Вы должны будете следить за системой, чтобы гарантировать ее правильную работу и соответствие потребностям предприятия. Вам придется наблюдать за действиями системы и выискивать ошибки, чтобы быстро исправлять их и предотвращать потерю доходов. И что особенно важно, вам понадобится выработать стратегию внедрения изменений, оказывающих влияние на работу сразу нескольких групп разработчиков, таких как изменение структуры предметного события.

Поддержка версий сообщений

В какой-то момент вам почти наверняка понадобится изменить формат некоторого сообщения. Но формат сообщения — это контракт, заключенный между двумя группами разработчиков. Любые изменения в формате могут потребовать организовать встречу групп и обсуждение изменений. Такие ситуации неизбежны, но одной из целей использования сервис-ориентированной архитектуры (Service-

Oriented Architecture, SOA) для создания слабосвязанных систем является устранение взаимовлияния групп, замедляющего их работу. Изменение контракта между несколькими ограниченными контекстами является той ситуацией, которая может привести к такому замедлению.

Если просто изменить формат сообщения и развернуть его, системы, не модернизированные для работы с новым форматом, могут просто перестать работать. Чтобы избежать этого, нужна стратегия, позволяющая изменять форматы сообщений так, чтобы не нарушить работоспособность существующих потребителей. В этом случае все заинтересованные группы могли бы перейти на новый формат, когда им будет это удобно, а до тех пор использовать старый формат. Именно поэтому следует стремиться к сохранению обратной совместимости, изменяя форматы сообщений.

Поддержка обратной совместимости сообщений

Поддержание обратной совместимости может стать важным шагом, гарантирующим сохранение интеграции между ограниченными контекстами и обеспечивающим независимость разработки каждого ограниченного контекста, без необходимости тратить массу времени на организацию встреч разработчиков и планирование одновременного перехода на новый формат при его изменении. Рассмотрим событие `OrderCreated`, которое было определено выше в этой главе:

```
public class OrderCreated
{
    public string OrderId { get; set; }
    public string UserId { get; set; }
    public string[] ProductIds { get; set; }
    public string ShippingTypeId { get; set; }
    public DateTime Timestamp { get; set; }
    public double Amount { get; set; }
}
```

Представьте, что владельцами предприятия было решено дать клиентам возможность указывать несколько адресов. Все конкуренты предоставляют такую возможность, поэтому предприятие также должно поддерживать ее, чтобы оставаться конкурентоспособным. Для добавления новой возможности нужно включить в событие `OrderCreated` поле `AddressId`. Для многих фреймворков обмена сообщениями это критическое изменение, в том смысле, что добавление нового поля в контракт нарушит работоспособность существующих подписчиков, не поддерживающих последнюю версию.

А теперь рассмотрим ситуацию внимательнее. На событие `OrderCreated` подписаны два ограниченных контекста, `Shipping` и `Billing`. Важно отметить, что ограниченный контекст `Shipping` знает новый адрес, поэтому ему можно передать заказ. Но должен ли это знать ограниченный контекст `Billing`? В некоторых предметных областях в этом нет необходимости, потому что ограниченный контекст `Billing` посылает счет клиенту по электронной почте. Даже если бы счет высы-

лался обычной почтой, на конверте должен указываться адрес для отправки счета, а не адрес доставки. Стоит ли тогда отвлекать разработчиков ограниченного контекста `Billing` от создания чего-то более важного, только чтобы они обеспечили прием сообщений в новом формате, который им вообще неинтересен? Во многих случаях не стоит, и поддержка обратной совместимости сообщений поможет избавить разработчиков от ненужных беспокойств и траты драгоценного времени на решение технических проблем.

Разные фреймворки обмена сообщениями могут отличаться средствами, позволяющими определять и изменять форматы сообщений. Далее вы узнаете, как реализовать обратную совместимость сообщений при использовании механизма версий во фреймворке `NServiceBus`, но описанные идеи легко можно распространить на любые другие фреймворки.

Обработка разных версий сообщений с помощью полиморфических обработчиков `NServiceBus`

В этом разделе будет реализована обратная совместимость для события `Order-Created`, благодаря которой ограниченный контекст `Shipping` сможет использовать новую версию сообщения, а ограниченный контекст `Billing` — прежнюю. Для этого достаточно выполнить всего четыре шага.

1. Создать новую версию сообщения, наследующую оригинальную версию.
2. Изменить обработчики, где требуется обрабатывать новую версию сообщения.
3. Добавить описание подписки на новое сообщение в файлы `App.config` всех подписчиков.
4. Изменить реализацию отправителя, чтобы он посылал новую версию сообщения.

Никаких изменений в службы, не использующие новую версию сообщения, вносить не требуется (они будут продолжать получать сообщения в старом формате).

В данном примере описанные шаги выражаются в создании события `Order-Created_V2`, наследующего событие `OrderCreated`, изменении реализации `Shipping.BusinessCustomers.ArrangeShipping` для обработки нового события и, наконец, реализации отправки события `OrderCreated_V2` из ограниченного контекста `Sales`.

Сначала добавьте событие `OrderCratedEvent_V2` в ту же папку, где находится оригинальное событие `OrderCreated`. Определение события должно выглядеть, как показано в листинге 12.19. Затем измените `OrderCreatedHandler`, как показано в листинге 12.20, чтобы использовать поле `AddressID`, появившееся в новой версии сообщения. Вам также понадобится изменить содержимое файла `App.config` в проекте `Shipping.BusinessCustomers.ShippingArranged`, как показано в листинге 12.21, чтобы указать, где будет публиковаться новое событие `OrderCreated_V2`. В заключение можно организовать отправку события `OrderCreated_V2`, как показано в листинге 12.22.

Листинг 12.19. Событие OrderCreated_V2

```
using System;

namespace Sales.Messages.Events
{
    public class OrderCreated_V2 : OrderCreated
    {
        public string AddressId { get; set; }
    }
}
```

Листинг 12.20. OrderCreatedHandler теперь обрабатывает событие OrderCreated_V2

```
// дополнение: обрабатывает сообщения версии V2
public class OrderCreatedHandler : IHandleMessages<OrderCreated_V2>
{
    // зависимость, внедряемая фреймворком NServiceBus
    public IBus Bus { get; set; }

    // дополнение: принимает сообщения версии V2
    public void Handle(OrderCreated_V2 message)
    {
        Console.WriteLine(
            "Shipping BC storing: Order: {0} User: {1} Address: {3}" +
            "Shipping Type: {2}",
            message.OrderId, message.UserId, message.ShippingTypeId,
            message.AddressId
        );
        var order = new ShippingOrder
        {
            UserId = message.UserId,
            OrderId = message.OrderId,
            AddressId = message.AddressId,
            ShippingTypeId = message.ShippingTypeId
        };
        ShippingDatabase.AddOrderDetails(order);
    }
}

public class PaymentAcceptedHandler : IHandleMessages<PaymentAccepted>
{
    // зависимость, внедряемая фреймворком NServiceBus
    public IBus Bus { get; set; }

    public void Handle(PaymentAccepted message)
    {
        var address = ShippingDatabase.GetCustomerAddress(
            message.OrderId
        );
        var confirmation = ShippingProvider.ArrangeShippingFor(
            address, message.OrderId
        );
        if (confirmation.Status == ShippingStatus.Success)
```

```

    {
        var evnt = new Shipping.Messages.Events.ShippingArranged
        {
            OrderId = message.OrderId
        };
        Bus.Publish(evnt);
        Console.WriteLine(
            "Shipping BC arranged shipping for Order: {0} to: {1}",
            message.OrderId, address
        );
    }
    else
    {
        // .. сообщить об ошибке организации доставки
    }
}

public static class ShippingDatabase
{
    private static List<ShippingOrder> Orders = new List<ShippingOrder>();

    public static void AddOrderDetails(ShippingOrder order)
    {
        Orders.Add(order);
    }

    public static string GetCustomerAddress(string orderId)
    {
        var order = Orders.Single(o => o.OrderId == orderId);

        // в действительности здесь вы могли бы организовать
        // поиск адреса клиента, сохраненного в ходе
        // обработки сообщения AddressCreated или аналогичного ему
        return string.Format(
            "{0}, Address ID: {1}",
            order.UserId, order.AddressId
        );
    }
}

public class ShippingOrder
{
    public string UserId { get; set; }

    public string OrderId { get; set; }

    public string ShippingTypeId { get; set; }

    public string AddressId { get; set; }
}

```

Листинг 12.21. Измененный раздел файла App.config, описывающий подписку на сообщение OrderCreated_V2

```
<UnicastBusConfig>
  <MessageEndpointMappings>
    <add Messages="Sales.Messages"
          Type="Sales.Messages.OrderCreated_V2"
          Endpoint="Sales.Orders.OrderCreated" />
    <add Messages="Billing.Messages"
          Type="Billing.Messages.PaymentAccepted"
          Endpoint="Billing.Payments.PaymentAccepted" />
  </MessageEndpointMappings>
</UnicastBusConfig>
```

Листинг 12.22. PlaceOrderHandler теперь посылает сообщение OrderCreated_V2

```
public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
{
    public IBus Bus { get; set; }

    public void Handle(PlaceOrder message)
    {
        var orderId = Database.SaveOrder(
            message.ProductIds, message.UserId,
            message.ShippingTypeId
        );

        Console.WriteLine(
            "Created order #{3} : Products:{0} with shipping: {1}" +
            " made by user: {2}",
            String.Join(", ", message.ProductIds),
            message.ShippingTypeId, message.UserId, orderId
        );

        // дополнение: теперь отправляет сообщение версии V2
        var orderCreatedEvent =
            new Sales.Messages.Events.OrderCreated_V2
        {
            OrderId = orderId,
            UserId = message.UserId,
            ProductIds = message.ProductIds,
            ShippingTypeId = message.ShippingTypeId,
            TimeStamp = DateTime.Now,
            Amount = CalculateCostOf(message.ProductIds),
            // Не бойтесь добавить сюда полную реализацию
            AddressId = "AddressID123"
        };

        Bus.Publish(orderCreatedEvent);
    }

    private double CalculateCostOf(IEnumerable<string> productIds)
    {

```

```
// поиск в базе данных и т. д.  
return 1000.00;  
}  
}
```

Если теперь запустить приложение, ограниченный контекст `Billing` должен действовать как прежде, обрабатывая сообщение в первоначальном формате (так как `OrderCreated_V2` содержит все свойства `OrderCreated`). Ограниченный контекст `Shipping` включает логику обработки нового поля с адресом, используя все тот же тип `OrderCreated` событий, но выполняя приведение к типу `OrderCreated_V2`. Важно отметить, что адрес доставки заказа не будет передаваться в событии `OrderCreated`. Он будет посылаться в составе другого события, такого как `UserAddedAddress`. Бизнес-компонент `Shipping.BusinessCustomers` должен подписаться на это событие и обеспечить сохранение адреса локально.

ПРИМЕЧАНИЕ

Дополнительные примеры полиморфической поддержки версий сообщений можно найти в документации к `NServiceBus` (<http://support.nservicebus.com/customer/portal/articles/894151-versioning-sample>).

Мониторинг и масштабирование

После передачи системы в эксплуатацию важно постоянно следить за появлением ошибок, чтобы гарантировать пользователям достаточно высокий уровень обслуживания. Также важно обеспечить оценку соответствия соглашениям об уровне обслуживания (`Service Level Agreement`, `SLA`) и экономических показателей, чтобы заинтересованные лица могли использовать их как основу для принятия важных решений, таких как выбор бизнес-модели или направления дальнейшего развития продукта. Оценка соглашений об уровне обслуживания сообщит разработчикам, какие участки системы работают слишком медленно и когда наступит момент приступить к решению вопроса масштабирования. Далее вы узнаете, как обрабатывать ошибки, как измерять технические и экономические показатели и как масштабировать ограниченные контексты с использованием `NServiceBus`. Эти идеи, безусловно, найдут применение при работе с другими фреймворками обмена сообщениями, хотя техническая их реализация, разумеется, будет выглядеть иначе.

Мониторинг ошибок

Ранее в этой главе вы видели, что для сообщений, доставка или обработка которых потерпела неудачу, выполняется повторная отправка. В том примере, однако, была добавлена дополнительная логика, гарантирующая лишь двукратный отказ фиктивной службы платежной системы. Такие ошибки называют неустойчивыми, или нерегулярными (`transient error`), потому что проблема решается сама собой с течением времени. Однако иногда ошибки носят систематический характер и, несмотря на количество повторений, попытки отправки сообщения всегда оканчиваются неудачей. Такие сообщения называют опасными (`poison messages`, подробное обсуждение опасных сообщений вы найдете по адресу <http://msdn>).

microsoft.com/ru-ru/library/ms166137.aspx). В некоторый момент фреймворк обмена сообщениями прекращает повторные попытки отправить опасное сообщение. Большинство фреймворков помещает их в специальную очередь с именем `error`, требующую ручного вмешательства для удаления сообщений.

Чтобы увидеть, как действует очередь `error`, можно изменить один из обработчиков сообщений, чтобы он всегда возбуждал исключение. Попробуйте, например, заменить логику метода `Handle()` в `Sales.Orders.OrderCreated.PlaceOrderHandler`, как показано ниже, чтобы симитировать систематическую ошибку:

```
public void Handle(PlaceOrder message)
{
    throw new Exception("I have received a poison message");
}
```

Если теперь запустить приложение и разместить заказ, это приведет к ошибке обработки сообщения и повторной его отправке несколько раз, после чего сообщение будет помещено в очередь `error` компонента, отправившего опасное сообщение (нужно подождать, пока в консоли появится сообщение об ошибке красного цвета.) Чтобы убедиться в этом, нужно открыть панель **Server Explorer** (Обозреватель сервера) в Visual Studio, распахнуть узел списка **Servers** (Серверы), распахнуть узел с именем вашего компьютера и затем распахнуть узел **Message Queues** (Очереди сообщений). После всего этого можно распахнуть узел **Private Queues** (Частные очереди) и заняться исследованием содержимого очереди `error`, как показано на рис. 12.26.

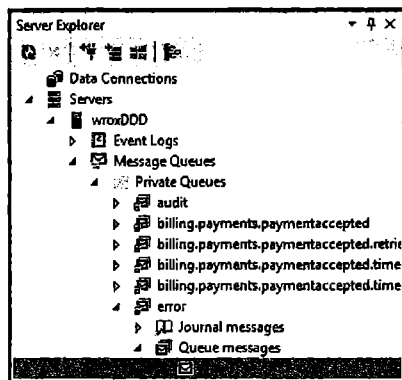


Рис. 12.26. Местонахождение очереди `error` в панели **Server Explorer** (Обозреватель сервера) в Visual Studio

Чтобы исследовать все сообщения в очереди `error`, распахните узел **Queue messages** (Сообщения в очереди). Чтобы увидеть содержимое отдельного сообщения, щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт **Properties** (Свойства), выделите строку **BodyStream** и щелкните на кнопке с тремя точками, которая появится в правой колонке. На рис. 12.27 показан пример просмотра значения **BodyStream** сообщения в очереди `error`.

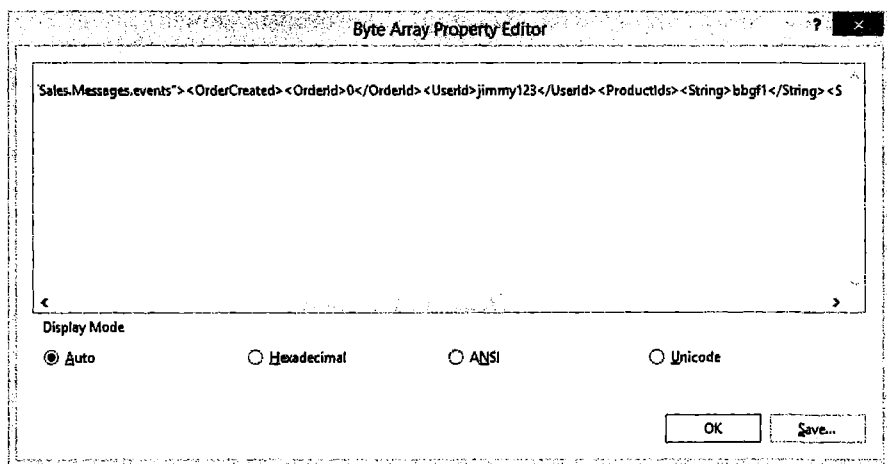


Рис. 12.27. Просмотр содержимого сообщения в очереди error

Мониторинг очереди error — самый важный вид деятельности, который поможет быстро обнаруживать ошибки в действующем программном обеспечении. Но как осуществлять мониторинг, во многом зависит от вас. К счастью, у вас на выбор есть множество вариантов. Например, можно использовать инструмент мониторинга Open Source Wolfpack (<http://wolfpack.codeplex.com/>) или даже создать собственную систему мониторинга.

Как только сообщение окажется в очереди error, необходимо ручное вмешательство для определения причины. После исследования сообщения может обнаружиться, что проблема решилась и сообщение может быть успешно обработано (если ошибка имела несистематический характер) или что ошибка находится в программном коде и сообщение никогда не будет благополучно обработано. В любом случае, когда проблема будет исправлена, можно воспользоваться инструментом `ReturnToSourceQueue.exe`, входящим в состав `NServiceBus` и предназначенным для отправки всех сообщений обратно в их очередь, чтобы повторить попытку обработки.

Дополнительную информацию о мониторинге и исправлении ошибок можно найти в документации к `NServiceBus` (<http://docs.particular.net/servicematrix/getting-started-with-nservicebus-using-servicematrix-2.0-fault-tolerance>).

Мониторинг соглашений об уровне обслуживания

Сколько заказов было размещено за последние два часа? Как долго обрабатывается один заказ? Как долго происходит списание средств с кредитной карты клиента внешней службой системы платежей? На все эти вопросы можно ответить, измеряя скорость и частоту сообщений, курсирующих в вашей системе. Эта информация важна для предприятия, так как помогает принимать обоснованные решения, она также важна для разработчиков, так как помогает выявлять узкие места и планировать оптимальное увеличение вычислительных мощностей.

Эти показатели можно измерять и передавать собственным аналитическим системам с применением выбранных технологий. Применяя MSMQ и NServiceBus, вы получаете множество превосходных и готовых к использованию инструментов мониторинга. Обе технологии поддерживают дополнительные счетчики производительности, полный перечень которых можно найти в документации к NServiceBus (<http://support.nservicebus.com/customer/portal/articles/859446-monitoring-nservicebus-endpoints>). Вы можете импортировать эти характеристики в свою систему мониторинга с помощью Windows Management Instrumentation ([http://msdn.microsoft.com/en-us/library/aa310909\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa310909(v=vs.71).aspx)).

Горизонтальное масштабирование

Если по окончании измерений характеристики говорят, что сообщения слишком много времени проводят в очереди, а работники предприятия — что клиенты жалуются на слишком медленную обработку заказов, вам следует выполнить расширение системы для ускорения ее работы. Так как покупать большие серверы экономически неэффективно, часто лучше добавить еще несколько серверов и распределить нагрузку между ними. Для системы передачи сообщений это означает, что одну и ту же очередь будет обслуживать несколько одинаковых компонентов. К сожалению, здесь кроется небольшая проблема, потому что NServiceBus создает очереди на машинах, которые обрабатывают сообщения, и не позволяет обращаться к очередям с других машин. Однако как раз для таких ситуаций в составе NServiceBus имеется решение балансировки нагрузки (load balancer), которое называется распределителем (distributor). Получить дополнительную информацию об этом решении можно в официальной документации к нему (<http://support.nservicebus.com/customer/portal/articles/859556-load-balancing-with-the-distributor>). Если появится нужда использовать другой фреймворк обмена сообщениями, он наверняка будет иметь похожее решение для масштабирования.

Прежде чем пытаться решить проблему добавления дополнительных аппаратных средств, можно попробовать изменить настройки NServiceBus, способные улучшить производительность фреймворка на единственной машине. Параметр `MaximumConcurrencyLevel` в разделе `TransportConfig`, как показано в следующем фрагменте, управляет числом потоков выполнения, используемых фреймворком NServiceBus. Следующий фрагмент настраивает выделение четырех потоков выполнения в распоряжение NServiceBus:

```
<TransportConfig MaxRetries="5" MaximumConcurrencyLevel="4" />
```

Интеграция ограниченных контекстов с применением MASS TRANSIT

Способ интеграции путем обмена сообщениями выбран потому, что он дает свободу выбора и независимость компонентам архитектуры. Однако вы оказываетесь привязаны к фреймворку NServiceBus, что, по сути, является еще одной формой зависимости (зависимости от платформы). Это может превратиться в проблему при создании новых систем с применением разных технологий или в том случае,

когда требуется интегрироваться с существующими системами, выполняющимися на платформах, отличных от .NET. Зависимость от платформы в системах обмена сообщениями — хорошо известная проблема с хорошо известным решением: шаблон с названием «Мост обмена сообщениями» (messaging bridge).

В этом разделе вы увидите, как использовать мост обмена сообщениями для интеграции с другой службой, использующей другой фреймворк обмена сообщениями — Mass Transit (мост обмена сообщениями также можно использовать для интеграции с системами, выполняющимися на платформах, отличных от .NET). Вы также узнаете немного об отличиях Mass Transit от NServiceBus и чуть больше — об особенностях работы с ним. Но сначала попробуем описать сценарий: компания, для которой вы создавали систему электронной коммерции выше в этой главе, приобрела успешный стартап, связанный с рекламой. Система проекта уже настроена на обработку сообщений и отправку бесплатных подарков случайным клиентам в момент размещения ими заказов. Чтобы интегрировать ее с приложением электронной коммерции на основе фреймворка NServiceBus, приобретенная система должна иметь возможность подписаться на события `OrderCreated`, посылаемые ограниченным контекстом `Sales`, как показано на рис. 12.28.

К сожалению, организовать такую подписку для нового ограниченного контекста `Promotions` оказалось непросто, потому что он не использует фреймворк NServiceBus. Решить эту проблему поможет мост обмена сообщениями.



Рис. 12.28. Интеграция ограниченного контекста Promotions с системой электронной коммерции

Мост обмена сообщениями

Когда имеется две отдельные системы, которые должны обмениваться сообщениями, один из способов решить проблему их интеграции заключается в использовании моста обмена сообщениями (<http://www.eaipatterns.com/MessagingBridge.html>). Обычно в таких ситуациях нет возможности сформировать из двух независимых фреймворков единую шину сообщений, но, создав мост обмена сообщениями, можно соединить конечные точки каждой системы сообщений и организовать связь между ними, как показано на рис. 12.29.

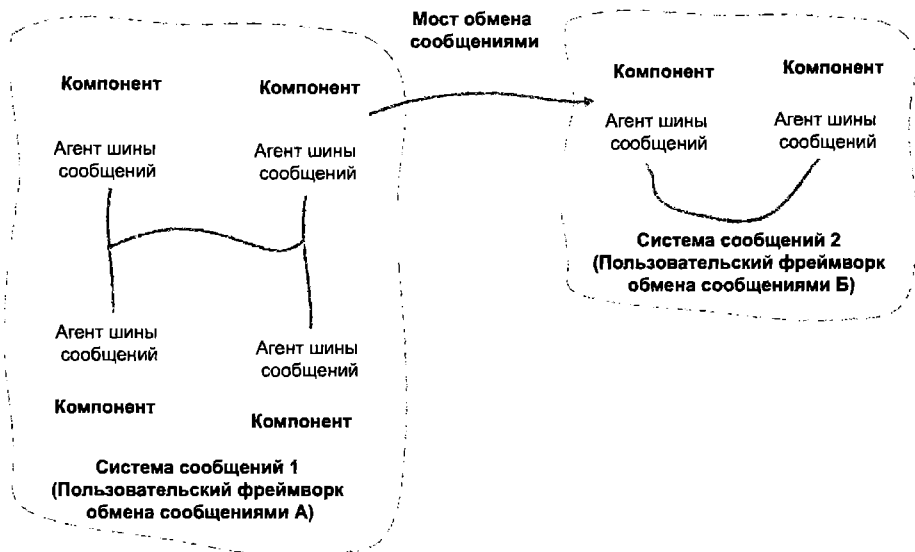


Рис. 12.29. Мост обмена сообщениями выполняет роль связного

Реализация моста обмена сообщениями обычно заключается в создании приложения, которое будет принимать сообщения из одной системы и передавать их в другую. Применительно к системе электронной коммерции это выражается в создании нового подписчика на события `OrderCreated_V2`, который будет преобразовывать их и передавать в новый ограниченный контекст, использующий фреймворк Mass Transit.

Но прежде чем приступить к его реализации, нужно создать ограниченный контекст `Promotions`, использующий Mass Transit. А перед этим — познакомиться с фреймворком Mass Transit.

Mass Transit

Часто разнообразие — это хорошо, именно поэтому можно считать, что разработчикам .NET повезло иметь в своем распоряжении Mass Transit (<http://masstransit-project.com/>) — фреймворк обмена сообщениями, являющийся достойной альтернативой `NServiceBus`, который можно использовать отдельно или в сочетании

с другими фреймворками обмена сообщениями. Последнее как раз и будет продемонстрировано в данном примере. Несмотря на то что он имеет отличающуюся реализацию и собственный прикладной программный интерфейс, Mass Transit имеет много общего с NServiceBus.

Установка и настройка Mass Transit

В существующее решение с реализацией системы на основе NServiceBus нужно добавить новую библиотеку классов C# с именем Promotions.LuckyWinner.LuckyWinnerSelected. Затем в нее можно будет добавить Mass Transit, выполнив следующие команды в консоли диспетчера пакетов (каждая должна вводиться в одну строку):

```
Install-Package MassTransit
-ProjectName Promotions.LuckyWinner.LuckyWinnerSelected
- Version 2.9.5
```

и

```
Install-Package MassTransit.msmq
-ProjectName Promotions.LuckyWinner.LuckyWinnerSelected
- Version 2.9.5
```

Теперь в проекте имеются все необходимые ссылки на Mass Transit, поэтому следующим нашим шагом будет настройка экземпляра шины Mass Transit. Вам наверняка будет приятно узнать, что все настройки Mass Transit можно выполнить прямо в программном коде. Их совсем немного, как показано в листинге 12.23. Добавьте содержимое листинга 12.23 в свой проект.

Листинг 12.23. Минимальные настройки Mass Transit в программном коде

```
using MassTransit;
using System;
using System.Linq;

namespace Promotions.LuckyWinner.LuckyWinnerSelected
{
    class Program
    {
        static void Main(string[] args)
        {
            Bus.Initialize(config =>
            {
                config.UseMsmq();
            });
        }
    }
}
```

Сначала выполняется импортирование классов Mass Transit (первая инструкция using). В методе Main выполняется настройка шины вызовом метода Bus.

`Initialize()`, которому передается объект с настройками, соответствующими вашим потребностям. В данном примере шина настраивается на использование службы очередей сообщений MSMQ, с которой вы познакомились выше в этой главе.

Несмотря на то что код в листинге 12.23 демонстрирует минимально необходимые настройки для использования службы MSMQ, он ничего не сообщает фреймворку Mass Transit о типах сообщений, которые ему предстоит обрабатывать, так же как он ничего не сообщает о том, как обрабатывать эти сообщения.

Объявление сообщений для Mass Transit

Следуя минималистскому стилю повествования, отметим, что сообщениями в Mass Transit могут быть любые классы C#; они не требуют именовать их определенным образом и не должны наследовать никаких других классов. Так как в этом примере создается мост обмена сообщениями с `NServiceBus`, можно просто добавить ссылку и использовать класс `OrderCreated`, который публикуется фреймворком `NServiceBus`. Однако чтобы сделать пример независимым, лучше создать копию класса в проекте. При таком подходе две стороны моста не будут связаны зависимостями. Кроме того, при создании кроссплатформенного моста обмена сообщениями такая ссылка все равно была бы невозможной.

Ниже приводится версия класса `Program` из листинга 12.23, дополненная определением класса `OrderCreated`, представляющего предметное событие создания заказа. Теперь этот класс будет доступен фреймворку Mass Transit; вам лишь нужно сообщить ему, как и когда его использовать. В листинге 12.24 показано, как должно выглядеть определение события `OrderCreated`.

Листинг 12.24. Событие `OrderCreated` для использования фреймворком Mass Transit

```
namespace Promotions.LuckyWinner.LuckyWinnerSelected
{
    class Program
    {
        // то же, что и в предыдущем листинге
        ...
    }

    public class OrderCreated
    {
        public string OrderId { get; set; }

        public string UserId { get; set; }

        public List<string> ProductIds { get; set; }

        public string ShippingTypeId { get; set; }

        public DateTime TimeStamp { get; set; }

        public double Amount { get; set; }
    }
}
```

Создание обработчика сообщений

И снова Mass Transit предлагает упрощенный, исключительно программный способ настройки обработчиков событий. Чтобы создать обработчик событий, достаточно где-нибудь определить метод, который будет обрабатывать сообщение. После этого можно сообщить фреймворку Mass Transit, что он должен вызывать этот метод при получении сообщения указанного типа. Вы поймете это после того, как увидите пример.

Создайте класс `OrderCreatedHandler` с единственным методом `Handle()`, принимающим экземпляр только что созданного события `OrderCreated`. Для простоты поместите определение класса в тот же файл, ниже класса `Program`, как это было проделано с классом `OrderCreated`. Реализация класса `OrderCreatedHandler` приводится в листинге 12.25.

Листинг 12.25. Класс `OrderCreateHandler` для проекта, использующего Mass Transit

```
public class OrderCreatedHandler
{
    public void Handle(OrderCreated message)
    {
        Console.WriteLine(
            "Mass Transit handling order placed event: Order:" +
            " {0} for User: {1}",
            message.OrderId, message.UserId
        );
    }
}
```

Теперь вы готовы сообщить фреймворку Mass Transit, как связать событие и его обработчик.

Подписка на события

Чтобы добавить подписку на событие с Mass Transit, нужно добавить совсем немного программного кода, информирующего фреймворк о том, какую очередь он должен просматривать в поисках сообщений и как с ними следует поступить. В листинге 12.26 приводится дополненная версия инициализации шины из листинга 12.23, куда добавлен код, реализующий подписку на событие `OrderPlaced`.

Листинг 12.26. Добавление подписки в настройки Mass Transit

```
static void Main(string[] args)
{
    Bus.Initialize(config =>
    {
        config.UseMsmq();
        // просматривать эту очередь в поисках событий размещения заказа
        config.ReceiveFrom("msmq://localhost/promotions.ordercreated");
        // подписаться на события размещения заказа
        config.Subscribe(sub =>
```

```
{
    // обрабатывать события размещения заказа, как показано здесь
    sub.Handler<OrderCreated>(msg =>
        new OrderCreatedHandler().Handle(msg)
    );
});
// чтобы окно консоли не закрывалось
while(true)
{
    Thread.Sleep(1000);
}
}
```

Первая новая строка сообщает фреймворку Mass Transit, что он должен просматривать очередь `promotions.ordercreated` на машине, где действует приложение (`localhost`). Mass Transit автоматически создаст эту очередь, и она станет доступна в панели **Server Explorer** (Обозреватель серверов) в Visual Studio, как было показано выше, когда обсуждалась очередь `error`. Когда Mass Transit обнаружит сообщение в этой очереди, он найдет подписку для данного типа сообщений. Как настраивается обработка события `OrderCreated` в такой подписке, можно видеть в новых строках, в листинге 12.26. Вызов метода `config.Subscribe()` просто сообщает фреймворку Mass Transit, что он должен создать новый экземпляр `OrderPlacedHandler` и передать сообщение, извлеченное из очереди, его методу `Handle`.

Соединение систем мостом обмена сообщениями

Теперь самая интересная часть: реализация моста обмена сообщениями. Для этого в данном примере нужно создать новый обработчик `NServiceBus`, подписанный на событие `NServiceBus` и передающий его в очередь Mass Transit, настроенную на прием. Представьте, что вы не используете ни Mass Transit, ни .NET, ни Windows. В действительности это не имеет никакого значения, потому что вы вручную передаете сообщение в очередь, откуда другой фреймворк обмена сообщениями, выполняющийся в другой среде или операционной системе, сможет извлечь его.

ПРИМЕЧАНИЕ

Мост обмена сообщениями — это лишь шаблон соединения двух отличающихся систем обмена сообщениями. А так как это всего лишь шаблон, он не имеет эталонной реализации. Реализовать мост обмена сообщениями можно самыми разными способами. Например, можно использовать простые файлы или базу данных, как было описано в предыдущей главе.

Начнем строительство моста обмена сообщениями с создания новой библиотеки классов C# с именем `Promotions.LuckyWinner.LuckyWinnerSelected.Bridge` внутри того же решения, куда добавим класс `OrderCreatedHandler`, представленный в листинге 12.27. Вам понадобится импортировать `NServiceBus` с использовани-

ем диспетчера пакетов, настроить соглашения о сообщениях, добавить ссылку на `Sales.Messages` и настроить подписку в `App.config`. (Чтобы вспомнить, как все это делается, обращайтесь к предыдущим разделам этой главы или загляните в файлы с примерами к этой главе.)

Листинг 12.27. Класс `OrderCreatedHandler` в реализации моста обмена сообщениями

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NServiceBus;
using System.Messaging;
using Sales.Messages.Events;
using System.Xml.Linq;
using System.Xml;

namespace Promotions.LuckyWinner.LuckyWinnerSelected.Bridge
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated_V2>
    {
        public void Handle(OrderCreated_V2 message)
        {
            Console.WriteLine(
                "Bridge received order: {0}. " +
                "About to push it onto Mass Transit's queue",
                message.OrderId
            );
            var massMsg = ConvertToMassTransitXmlMessageFormat(message);
            var msmqMsg = new Message
            {
                Body = XDocument.Parse(massMsg).Root,
                Extension = Encoding.UTF8.GetBytes(
                    "{ \"Content-Type\": \"application/vnd.masstransit+xml\" }"
                )
            };
            var queue = new MessageQueue(
                ".\\Private$\\promotions.ordercreated", QueueAccessMode.Send
            );
            queue.Send(msmqMsg);
        }

        // в промышленной версии используйте более надежную стратегию
        // данный подход применяется, только чтобы подчеркнуть XML-формат,
        // требуемый для работы с Mass Transit
        private string ConvertToMassTransitXmlMessageFormat(
            OrderCreated_V2 message)
        {
            return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
                "<envelope>" +
                "<headers />" +
```

```

        "<destinationAddress>" +
"msmq://localhost/"promotions.ordercreated?tx=false&recoverable=true" +
        "</destinationAddress>" +
        "<message>" +
            "<orderId>" + message.OrderId + "</orderId>" +
            "<userId>" + message.UserId + "</userId>" +
            GenerateProductIdsXml(message.ProductIds) +
            "<shippingTypeId>" + message.ShippingTypeId +
            "</shippingTypeId>" +
            "<amount>" + message.Amount + "</amount>" +
            "<timestamp>" +
                XmlConvert.ToString(
                    message.TimeStamp,
                    XmlDateTimeSerializationMode.Utc
                ) +
            "</timestamp>" +
            "</message>" +
            "<messageType>" +
"urn:message:Promotions.LuckyWinner.LuckyWinnerSelected:OrderCreated" +
            "</messageType>" +
        "</envelope>";
    }

    private string GenerateProductIdsXml(IEnumerable<string> productIds)
    {
        return String.Join(
            "", productIds.Select(p => "<productIds>" + p + "</productIds>")
        );
    }
}
}
}

```

Как упоминалось выше, этот обработчик должен быть подписан на событие `NServiceBus` и передавать это событие в очередь `Mass Transit`. Здесь есть несколько тонкостей, которые вы должны знать и понимать. Во-первых, обратите внимание, что здесь отсутствуют какие-либо ссылки на `Mass Transit`. Все, что делает мост, — использует классы `MSMQ` из библиотеки `System.Messaging.dll` (вы должны добавить ссылку на нее) для передачи сообщений в очередь. Вторым важным аспектом, который следует учесть, является формат сообщений. `Mass Transit` сначала просматривает заголовок, чтобы определить тип содержимого в сообщениях. Вы можете видеть, что обработчик устанавливает тип `application/vnd.masstransit+xml`, сообщая фреймворку `Mass Transit`, что тело содержит разметку XML. Затем `Mass Transit` интерпретирует тело сообщения, ожидая, что его формат будет соответствовать формату, возвращаемому методом `ConvertToMassTransitXmlMessageFormat()`. После всего этого просто создается очередь и в нее помещается сообщение, как показано в двух последних строках метода `Handle()`.

В этом примере можно заметить несколько проблем, имеющих отношение к мосту обмена сообщениями. Чтобы передать сообщение другому фреймворку, нужно знать, какие форматы он понимает. Также нужно опуститься несколькими

уровнями абстракций ниже и работать с очередями вручную. Для изучения внутренних особенностей фреймворка может потребоваться дополнительное время, но эти знания никогда не будут лишними, особенно если вам часто приходится проводить мониторинг очередей и проверять, насколько правильно все работает.

Если вы считаете, что конструирование мостов сообщений является неэффективным решением для вашего проекта, но вам все еще необходимо обеспечить масштабируемость и отказоустойчивость с применением некоторых принципов реактивного программирования, тогда, возможно, вам пригодится архитектура REST. Данная архитектура и другие решения на основе HTTP рассматриваются в следующей главе.

Публикация событий

Публикация событий в Mass Transit выполняется почти так же, как в NServiceBus, как показано в следующем фрагменте:

```
Bus.Instance.Publish(new LuckyWinnerSelected(UserId = "user123"));
```

В качестве самостоятельного упражнения попробуйте создать на основе Mass Transit новый компонент, обрабатывающий события, публикуемые компонентом `Promotions.LuckyWinner.LuckyWinnerSelected`.

Тестирование

Теперь ваш мост сообщений должен работать, а ограниченный контекст `Promotions` — получить возможность обрабатывать события `OrderCreated`. Вам осталось только убедиться в этом. Чтобы протестировать приложение, выполните следующие шаги.

1. Настройте запуск проекта и приложение консоли, следуя рекомендациям в этой статье: <http://possiblythemostboringblogever.blogspot.co.uk/2012/08/creating-console-app-in-visual-studio.html>.
2. Запустите систему (нажав F5).
3. Создайте заказ, как описывалось выше (перейдя по адресу `/orders`).
4. Проверьте, выводятся ли в консолях NServiceBus сообщения, как прежде.
5. Проверьте прием события `OrderCreated` в консоли для компонента Mass Transit. Если все работает как надо, содержимое в окне консоли должно напоминать рис. 12.30. (У вас будут отображаться другие идентификаторы пользователей, которые вы ввели в форму.)

```
Mass Transit handling order placed event: Order: 0 for User: Wrox  
Mass Transit handling order placed event: Order: 1 for User: Scott  
Mass Transit handling order placed event: Order: 2 for User: Nick
```

Рис. 12.30. Mass Transit принимает сообщения через мост

Где найти дополнительную информацию о Mass Transit

Mass Transit обладает множеством полезных функциональных возможностей, которые не были показаны в этой главе. Например, его можно использовать с другими технологиями организации очередей, такими как Rabbit MQ. Существует также весьма дружное сообщество пользователей Mass Transit. Поэтому если у вас появится сильное желание узнать больше об этом фреймворке, вы сможете заглянуть в его исходные тексты, доступные на GitHub (<https://github.com/phantboyg/MassTransit>), ознакомиться с официальной документацией (<http://masstransit-project.com/>), а также почитать или задать свои вопросы в списке рассылки Mass Transit (<https://groups.google.com/forum/#!forum/masstransit-discuss>).

Ключевые идеи

- Системы обмена сообщениями используются для создания масштабируемых, отказоустойчивых систем, основанных на принципах реактивного программирования и слабой связанности компонентов.
- Применяя принципы философии предметно-ориентированного проектирования (Domain-Driven Design, DDD) при создании систем обмена сообщениями, используйте в роли сообщений предметные события, чтобы явно отразить предметные понятия в программном коде.
- Диаграммы контейнеров помогут вам определить архитектуру системы обмена сообщениями.
- Диаграммы компонентов можно использовать для моделирования потоков сообщений с использованием единого языка.
- Команды — это сообщения, определяющие, что должно произойти; они обрабатываются в одном месте.
- События используются, чтобы сообщить о происшедшем; они могут обрабатываться множеством подписчиков.
- Шина сообщений — это распределенный компонент, имеющий агентов, выполняющихся внутри каждого приложения, которое должно иметь возможность посылать и/или принимать сообщения.
- Мост обмена сообщениями — это шаблон программирования, используемый для сокрытия сложностей, связанных со взаимодействиями с внешними системами, и увеличения надежности.
- При создании систем обмена сообщениями часто дополнительно используется принцип локального хранения данных для большей масштабируемости и отказоустойчивости. Но при этом возникает эффект потенциальной непротиворечивости.
- Потенциальная непротиворечивость часто является необходимым условием эффективной масштабируемости.

- Движение вперед, к новым состояниям, — это типичный шаблон решения проблем, связанных с потенциальной непротиворечивостью, которая может привести к открытию новых возможностей.
- Наполнение пользовательского интерфейса с применением прикладных программных интерфейсов, которые импортируются компонентами, считается рекомендуемой практикой представления информации в веб-приложениях.
- Поддержание обратной совместимости версий сообщений является ключевым аспектом, гарантирующим, что группы разработчиков не будут тормозить друг друга, когда им потребуется изменить форматы сообщений.
- Для соединения несовместимых технологий обмена сообщениями можно использовать мосты обмена сообщениями, но это не всегда самое лучшее или надежное решение; альтернативы на основе HTTP, такие как REST, могут оказаться предпочтительнее.

13

Интеграция с RPC и REST посредством HTTP

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Знакомство с особенностями использования HTTP для интеграции ограниченных контекстов
- Введение в протокол REST
- Рекомендации по выбору между RPC и REST для интеграции посредством HTTP
- Примеры применения DDD для реализации RPC с использованием SOAP и XML
- Примеры реализации RPC с использованием WCF и ASP.NET Web API
- Обсуждение вопросов использования REST с DDD для увеличения отказоустойчивости и масштабируемости систем обмена сообщениями с сохранением предметной области в центре внимания
- Пример создания масштабируемой, отказоустойчивой, управляемой событиями распределенной системы RESTful с использованием ASP.NET Web API
- Рекомендации по поддержанию независимости групп разработчиков при интеграции ограниченных контекстов посредством HTTP

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 13 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Протокол передачи гипертекста (Hypertext Transport Protocol, HTTP) — это универсальный протокол, понятный миллиардам устройств, подключенных к Интернету. Он с успехом может использоваться для интеграции ограниченных контекстов. Будучи столь распространенным, протокол HTTP ясно доказал, что позволяет

относительно легко организовать взаимодействия между приложениями, которые выполняются на разных аппаратных и программных платформах. Таким образом, при наличии ограниченных контекстов, использующих разные технологии, протокол HTTP может оказаться весьма привлекательным выбором. В предыдущей главе было показано, что, несмотря на возможность интеграции разных фреймворков обмена сообщениями, на решение этой задачи может потребоваться значительное время. С другой стороны, благодаря широкой известности стандарта HTTP, вы можете следовать существующим соглашениям при интеграции с ним.

В главе 11 «Введение в интеграцию ограниченных контекстов» и главе 12 «Интеграция посредством обмена сообщениями» было показано, что интеграция ограниченных контекстов заключается не только в том, чтобы наладить взаимодействие между приложениями. Она также обеспечивает масштабируемость и отказоустойчивость. У кого-то из вас может возникнуть вопрос, почему было создано так много фреймворков обмена сообщениями и промежуточных решений, если протокол HTTP удовлетворяет все эти потребности. На этот вопрос трудно дать однозначный ответ, но это лишь подчеркивает тот факт, что HTTP часто незаслуженно упускают из виду при проектировании распределенных систем, управляемых событиями. Однако в последние годы все большую популярность приобретает архитектура REST как инструмент создания распределенных систем и ее определенно стоит рассматривать в числе других альтернатив. Эта глава расскажет вам почему.

Протокол HTTP редко применяется для создания реактивных систем, управляемых событиями, но он пользуется большой популярностью как средство интеграции приложений с использованием технологии вызова удаленных процедур (Remote Procedure Call, RPC). В противоположность архитектуре REST, существуют тысячи примеров приложений, использующих RPC для интеграции. Это означает, что существует огромное число свидетелей, демонстрирующих сильные и слабые стороны данной технологии. О некоторых из них вы узнали в главе 11, а в этой главе вы познакомитесь с конкретными примерами, создавая и сравнивая системы на основе RPC и RESTful с применением философии DDD.

Какое бы решение на основе HTTP вы ни выбрали, всегда найдутся шаблоны и принципы, которые в сочетании с приемами DDD помогут сделать предметные понятия более явными. Например, в главах 11 и 12 демонстрировалось, как переход на архитектуру, управляемую событиями и основанную на предметных событиях, может дать дополнительные экономические и технические выгоды. В этой главе вы узнаете, что использование предметных событий в качестве сообщений, передаваемых между ограниченными контекстами посредством REST, также может быть весьма выразительным.

Однако не всегда есть смысл использовать события как формат прикладного программного интерфейса на основе HTTP, особенно когда доступ к предметной области экспортируется как API к внешним службам. Представьте API, обеспечивающий доступ к каталогу товаров: веб-сайтам не нужна полная история событий, им нужна лишь самая последняя информация. Поэтому в данной главе также будут представлены примеры экспортирования предметных понятий в виде HTTP API, не являющихся событиями.

ПРИМЕЧАНИЕ

В этой главе рассматривается множество тем, но здесь недостаточно места для обсуждения их во всех подробностях. Архитектуре REST, например, посвящены целые книги. Однако в этой главе дается достаточный объем теории, примеров и описаний разных технологий, чтобы вы могли почувствовать себя готовыми к созданию разного вида систем, представленных в этой главе. И, как обычно, вы можете задавать любые вопросы или делиться своими мыслями на дискуссионном форуме Wrox, на сайте <http://p2p.wrox.com/>.

В конце этой главы мы обсудим возможность поддержания независимости групп разработчиков для увеличения эффективности разработки ими своих ограниченных контекстов. В главе 11 в общих чертах было показано, как некоторые идеи, такие как сервис-ориентированные архитектуры (Service-Oriented Architecture, SOA), поддерживают такую возможность, а в главе 12 — как воплотить эти идеи в реализацию архитектуры обмена сообщениями. В этой главе будут даны аналогичные рекомендации для поддержания независимости групп разработчиков, использующих HTTP в качестве протокола интеграции.

Преимущества HTTP

Если каждое устройство с доступом к сети Интернет использует протокол HTTP, тому должны быть веские основания. Ниже перечислены некоторые причины, объясняющие, почему интеграция ограниченных контекстов посредством HTTP может оказать решающее влияние на успех проекта, к разработке которого вы привлечены.

Независимость от выбора платформы

Компоненты и приложения, пригодные для интеграции посредством HTTP, можно создавать с использованием любых технологий благодаря независимости протокола от платформы. Это не только благоприятно для создания слабосвязанных приложений, но также помогает организовывать группы разработчиков, независимых друг от друга.

При использовании протокола HTTP ограниченные контексты должны придерживаться установленных общих контрактов, в число которых входят форматы HTTP-запросов и ответов. При соблюдении контрактов группы разработчиков могут свободно реструктурировать свои приложения, переписывать их с применением новых технологий или продолжать добавлять новые функциональные возможности в своем ритме. Единственное, что имеет значение, — они должны придерживаться установленных общих контрактов, чтобы не нарушить интеграцию с другими ограниченными контекстами.

HTTP понятен любому

HTTP — вездесущий протокол. Почти все языки программирования и среды выполнения имеют обширные библиотеки поддержки HTTP. Поэтому инте-

грация посредством HTTP имеет мощную поддержку. В этой главе вы узнаете, что для платформы .NET имеется множество фреймворков, включая Windows Communication Foundation (WCF) и ASP.NET Web API, предназначенных для создания интегрированных решений на основе HTTP.

Широкая известность HTTP является еще одним преимуществом, которое становится особенно очевидным, когда приходит время расширять коллектив. Часто бывает трудно найти людей, знакомых с системами обмена сообщениями или конкретными фреймворками, но вы с легкостью найдете разработчиков, знающих и понимающих HTTP.

Множество проверенных инструментов и библиотек

Поверх современных фреймворков и библиотек для создания решений интеграции на основе HTTP создан ряд улучшенных инструментов. Одним из примеров может служить автоматическое создание в Visual Studio классов после выбора конкретного типа веб-службы. Эти классы содержат методы, имитирующие API веб-служб на основе HTTP, благодаря чему вы можете писать обычный объектно-ориентированный код, который тем не менее осуществляет взаимодействия с использованием сети. Это будет продемонстрировано ниже, в примерах использования WCF.

Возможность пользоваться своими же API

При организации взаимодействий посредством HTTP потребность иметь отдельные каналы для внутренних и внешних взаимодействий может отпасть сама собой. Выражаясь простым языком, это означает, что можно определить API, которыми будут пользоваться не только ограниченные контексты, но и третьи стороны. Системы обмена сообщениями, напротив, почти всегда используются исключительно для внутренних нужд, в связи с чем приходится конструировать еще и внешние API.

Практика использования прикладных программных интерфейсов (API) одновременно для внутренних нужд и обслуживания клиентов и партнеров известна под названием «dogfooding»¹. Такая практика весьма желательна, потому что помогает вовремя заметить проблемы, с которыми сталкиваются клиенты. Если API доставляет клиентам болезненные проблемы, «dogfooding» (прием одновременного использования API для внутренних и внешних взаимодействий) поможет увидеть и устранить их. Конечно, в некоторых ситуациях «dogfooding» доставляет больше хлопот, чем преимуществ, и является не самым лучшим подходом. Примером может служить ситуация, когда к скорости внутренних взаимодействий предъявляются более строгие требования, чем к скорости внешних взаимодействий.

¹ Термин «dogfooding» не имеет точного аналога в русском языке. Под ним подразумевается использование выпускаемого продукта (в данном случае экспортируемого прикладного программного интерфейса) для внутренних нужд, как говорилось в рекламе одной известной компании, выпускающей корм для собак: «Наша еда для собак настолько вкусная, что мы едим ее сами». — *Примеч. пер.*

RPC

При создании распределенных систем, внутренние контексты в которых интегрируются через HTTP, есть возможность использовать RPC. Как обсуждалось в главе 11, механизм RPC «скрывает факт сетевых взаимодействий», что может оказаться решающим фактором, когда важна высокая скорость разработки или почти отсутствует необходимость в масштабируемости. С другой стороны, образование тесных связей между компонентами, что так свойственно RPC, может существенно затруднить достижение независимости групп разработчиков или соответствие требованиям масштабируемости.

В следующем разделе приводятся примеры, демонстрирующие сильные и слабые стороны RPC, о которых упоминалось ранее. Они помогут вам сформировать свое мнение и представление о том, где бы вы смогли использовать RPC в будущем.

ПРИМЕЧАНИЕ

Многие компании, даже с весьма строгими требованиями к масштабируемости, все еще используют RPC посредством HTTP в качестве основной стратегии интеграции, что говорит о неплохом потенциале этого механизма. Однако по мере увеличения численности группы работа над проектом все больше начинает напоминать тушение пожара, нежели спокойную разработку новых особенностей. Многие из них, достигнув определенного уровня, начинают переходить на асинхронные и управляемые событиями альтернативы.

Реализация RPC через HTTP

Определившись с решением реализовать RPC посредством HTTP, вы имеете на выбор несколько вариантов. Традиционный вариант основан на использовании простого протокола доступа к объектам (Simple Object Access Protocol, SOAP), добавляющего еще один уровень абстракции поверх HTTP. В последние годы, однако, наблюдается значительное падение популярности SOAP. В настоящее время механизмы RPC чаще реализуют на основе передачи данных в виде текста на расширяемом языке разметки (eXtensible Markup Language, XML) или в формате представления объектов JavaScript (JavaScript Object Notation, JSON). Поэтому в данном разделе будут представлены оба варианта, начиная с SOAP, чтобы вы могли принимать обоснованные решения.

ПРИМЕЧАНИЕ

Чтобы опробовать примеры из этой главы, понадобится установить бесплатную версию среды разработки Visual Studio Community. Загрузить ее можно на сайте <https://www.visualstudio.com/products/free-developer-offers-vs.aspx>. Также отлично подойдут полные версии Visual Studio.

SOAP

SOAP в полной мере соответствует понятию RPC, включая в сообщения массу информации, такой как тип и метаданные функции. Это упрощает преобразование содержимого сообщений SOAP в вызовы методов на удаленном сервере. Именно

богатство передаваемой информации обеспечило протоколу SOAP огромную популярность у предыдущего поколения разработчиков, благодаря которой, в свою очередь, было создано множество замечательных инструментов поддержки SOAP, в чем вы вскоре убедитесь.

Чтобы поближе познакомиться с SOAP и опробовать инструменты поддержки, в этом разделе мы реализуем интеграцию двух ограниченных контекстов, формирующих часть приложения социальной сети. Представьте, что вы участвуете в разработке продукта наподобие Twitter, который находит все большую поддержку среди все возрастающего числа пользователей.

Чтобы помочь группам разработчиков быстрее двигаться вперед, вы решили использовать RPC. В настоящее время они имеют единое, монолитное приложение, напоминающее «большой ком грязи», в котором ограниченные контексты оформлены как простые библиотеки, тесно связанные друг с другом на уровне двоичного кода. Это обстоятельство вызывает множество проблем, когда изменения в одном ограниченном контексте нарушают работу других. Вы собираетесь решить проблему, изолировав каждый ограниченный контекст и оформив их в виде отдельных приложений, взаимодействующих друг с другом только посредством HTTP. Чтобы сделать такой переход менее болезненным и быстрым, вы решили с помощью RPC заменить внутренние вызовы методов из одного контекста в другой вызовами удаленных процедур через сеть (полностью устранив зависимость на уровне двоичного кода). В этом примере вы увидите, что RPC требует лишь незначительной переделки программного кода и делает сетевые взаимодействия почти невидимыми.

Подготовка к переходу на использование RPC

Подготовка к переходу на использование RPC включает выбор методов, которые будут вызываться средствами RPC по сети. В остальном, кроме этих вызовов, программный код будет выглядеть так, как будто он выполняется на одной машине. На рис. 13.1 изображена новая архитектура текущего сценария. Обратите внимание, как вызовы методов между ограниченными контекстами замещаются в ней сетевыми операциями RPC.

В текущей (воображаемой) системе ограниченный контекст поиска вызывает метод `FindUsersFollowers()` класса `FollowerDirectory`, который принадлежит ограниченному контексту управления учетными записями пользователей. Это зависимость на уровне двоичного кода, требующая, чтобы взаимодействие протекало в рамках одного процесса. В листинге 13.1 показано, как осуществляется это взаимодействие.

Листинг 13.1. Текущее решение с зависимостью между ограниченными контекстами на уровне двоичного кода

```
namespace Discovery.Recommendations
{
    public class Recommender
    {
        public static List<Accounts> FindRecommendedUsers(string accountId)
        {
```



```

var fd = AccountManageent.FollowerDirectory;

// вызов другого ограниченного контекста в том же процессе
var followers = fd.FindUsersFollowers(accountId);

var recommendations = ApplyCleverRecommendationsAlgorithm(followers);

return recommendations;
    }
}
}
}

```

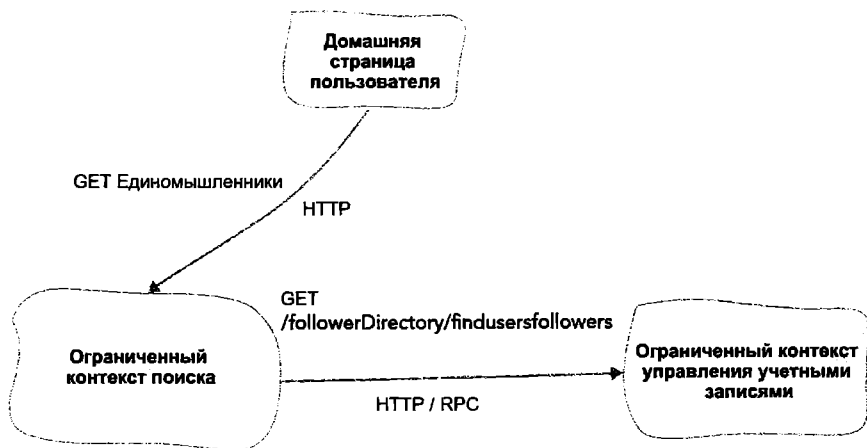


Рис. 13.1. Сценарий использования «Поиск возможных единомышленников»

В листинге 13.1 демонстрируется вызов метода `FollowerDirectory.FindUsersFollowers()`. Этот проблематичный метод образует тесную связь между двумя ограниченными контекстами. Его следует заменить аналогичным методом RPC с тем же именем, вызываемым по сети, благодаря чему будет устранена зависимость на уровне двоичного кода между ограниченными контекстами поиска и управления учетными записями.

Рисунок 13.1 также содержит дополнительную информацию о случае использования, который предполагается реализовать. Как можно видеть на рисунке, случай использования запускается после того, как пользователь регистрируется в системе и откроет свою домашнюю страницу. Когда это произойдет, на странице, согласно требованиям бизнеса, должен появиться список единомышленников, за которыми может последовать¹ данный пользователь. Это важно для бизнеса, потому что помогает пользователям находить возможных единомышленников и интересующие их темы, а это гарантирует, что пользователь будет возвращаться на сайт снова и снова. Весь коллектив, сотрудники и разработчики, сосредоточен на оказании помощи пользователям в получении интересующей их информации и известен как отдел поиска, а ограниченный контекст поиска представляет данный раздел предметной области.

¹ «Последовать» в терминологии Twitter означает подписаться на получение новых сообщений, оставляемых выбранным пользователем. — *Примеч. пер.*

Реализация RPC через HTTP с применением WCF и SOAP

Интеграция двух ограниченных контекстов с применением SOAP на первом этапе может продвигаться очень быстро и относительно легко, если задействовать фреймворк Windows Communication Foundation (WCF), входящий в состав .NET. Вы убедитесь в этом сразу же, как только приступите к реализации сценария использования, изображенного на рис. 13.1.

ВНИМАНИЕ

Несмотря на то что в этой главе используется прием деления решения на проекты по ограниченным контекстам, для промышленной системы это может оказаться не самым удачным выбором. Обязательно обдумайте возможность отделения механизмов передачи информации от предметной логики (веб-приложений, обычных приложений, командной строки и т. д.).

Создание службы WCF

Для начала нужно создать новое решение Visual Studio, где будут размещаться все ограниченные контексты (данного примера использования SOAP). Этому решению можно дать имя PPPDDD.SOAP.SocialMedia. Сначала создадим ограниченный контекст управления учетными записями. Для этого создайте новый проект типа WCF Service Application с именем AccountManagement, как показано на рис. 13.2.

ПРИМЕЧАНИЕ

Причина, по которой первым создается ограниченный контекст управления учетными записями, состоит в том, что он экспортирует веб-службу, которая будет вызываться ограниченным контекстом поиска. Как вы убедитесь, порядок создания проектов играет важную роль, потому что Visual Studio может автоматически создавать все необходимые прокси-классы, упрощающие обращения к веб-службе, если веб-служба уже существует.

В прежнем, монолитном приложении, как было показано в листинге 13.1, существует класс с именем FollowerDirectory, имеющий метод FindUsersFollowers. Чтобы превратить вызовы этого метода в вызовы RPC, достаточно просто добавить службу WCF в корень проекта и дать ей имя FollowerDirectory. После этого можно будет использовать контракты службы WCF для объявления методов RPC, как показано в следующем разделе.

ПРИМЕЧАНИЕ

Чтобы упростить проверку работы примеров, настройте проект AccountManagement так, чтобы он всегда использовал порт 3100. Этот параметр настройки можно найти в свойствах проекта, на вкладке Web.

Контракты службы

После добавления службы WCF в корень проекта появятся два файла: FollowerDirectory.svc.cs и IFollowerDirectory.cs. Последний из них используется средой Visual Studio для автоматического создания контракта SOAP (с использо-

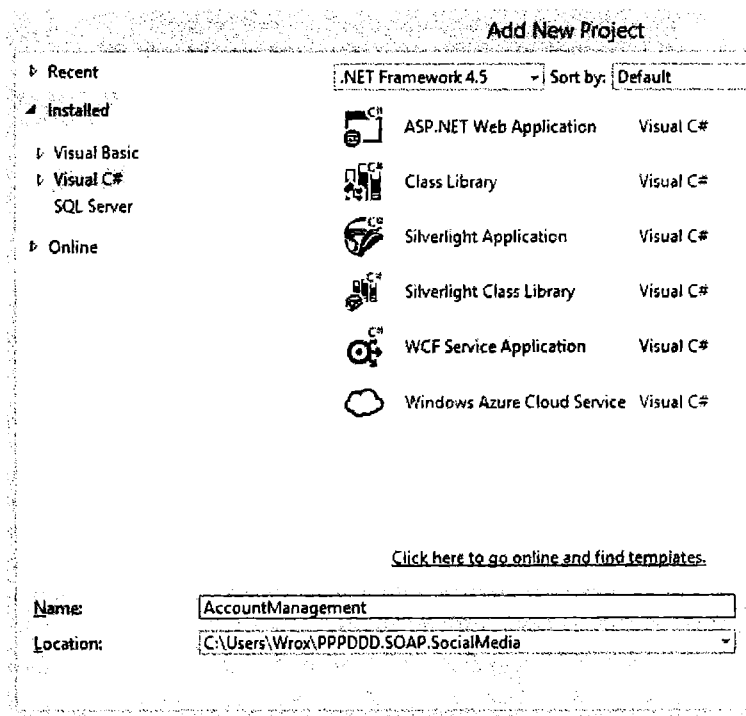


Рис. 13.2. Добавление службы WCF управления учетными записями

ванием языка описания веб-служб Web Service Description Language, или WSDL). В первом файле находится реализация; сюда вы должны помещать весь свой программный код, который должен выполняться при вызове методов RPC. Вскоре вы увидите, как он действует, поэтому не волнуйтесь, если что-то пока кажется вам непонятным.

Фреймворк WCF поддерживает две аннотации: `ServiceContract` и `OperationContract`. Аннотация `ServiceContract` добавляется к определению интерфейса, чтобы показать, что класс содержит методы, которые могут вызываться механизмом RPC по сети. Аннотация `OperationContract` отмечает методы RPC в интерфейсе, декорированном аннотацией `ServiceContract`. То есть, чтобы создать RPC-вызов `FollowerDirectory.FindUsersFollowers()`, необходимо применить эти две аннотации, как показано в листинге 13.2.

Листинг 13.2. Определение RPC-вызовов с помощью аннотации `OperationContract` в WCF

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace AccountManagement
```

```

{
    [ServiceContract] // сообщить WCF, что класс содержит методы RPC
    public interface IFollowerDirectory
    {
        [OperationContract] // сообщить WCF, что это – метод RPC
        List<Follower> FindUsersFollowers(string accountId);
    }

    public class Follower
    {
        public string FollowerId {get; set; }

        public string FollowerName { get; set;}

        public List<string> SocialTags { get; set; }
    }
}

```

Листинг 13.2 содержит все, что нужно среде Visual Studio и фреймворку WCF, чтобы сгенерировать инфраструктуру RPC. Далее вы узнаете, что на основе этих классов Visual Studio автоматически создает прокси-классы внутри клиентов веб-службы. Такая прозрачная поддержка сетевых взаимодействий делает применение WCF и SOAP особенно привлекательным.

Прежде чем новая служба заработает, нужно добавить реализацию в файл `FollowerDirectory.svc.cs`. В листинге 13.3 приводится простейшая реализация, которая генерирует несколько фиктивных единомышленников и возвращает их. Добавьте эту реализацию в свой проект `FollowerDirectory`.

Листинг 13.3. Простейшая реализация RPC-метода `FindUsersFollowers`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace AccountManagement
{
    public class FollowerDirectory : IFollowerDirectory
    {
        public List<Follower> FindUsersFollowers(string accountId)
        {
            return GenerateDummyFollowers().ToList();
        }

        private IEnumerable<Follower> GenerateDummyFollowers()
        {
            for (int i = 0; i < 10; i++)
            {
                yield return new Follower
                {
                    FollowerId = "follower_" + i,

```

```
FollowerName = "happy follower " + i,
SocialTags = new List<string>
{
    "programming", "DDD", "Psychology"
}
};
}
}
```

Тестирование служб WCF

Теперь все готово к проверке возможности вызова `FindUsersFollowers()` по сети посредством механизма `RPC`. `Visual Studio` упрощает эту задачу, предоставляя тестовый клиент. Чтобы запустить тестовый клиент, выберите элемент `FollowerDirectory.svc` в панели `Solution Explorer` (Обозреватель решения) — не `FollowerDirectory.sv.cs` — и нажмите клавишу `F5`. После этого откроется окно тестового клиента, как показано на рис. 13.3.

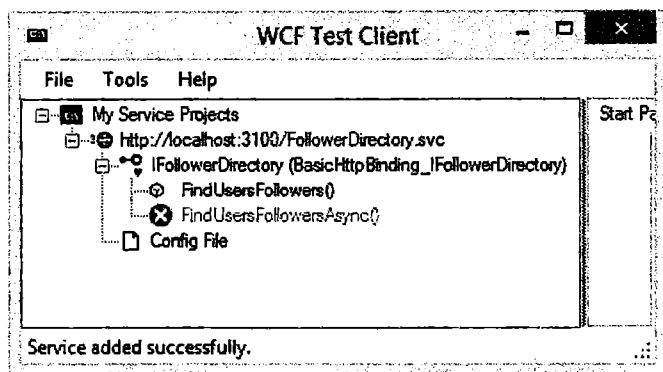


Рис. 13.3. Клиент Visual Studio для тестирования служб WCF

Чтобы протестировать новую службу, просто выполните двойной щелчок на ее имени (`FindUsersFollowers`) в панели Explorer (Обозреватель) слева и затем введите значение в столбце `Value` (Значение), в строке с параметром `accountId`, справа. Если после этого щелкнуть на кнопке `Invoke` (Вызвать), будет выполнен RPC-вызов службы `FindUsersFollowers` по сети и результаты появятся в нижней половине панели справа, как показано на рис. 13.4.

Если вам интересно узнать, какие данные были переданы по сети, щелкните на вкладке **XML** внизу панели справа. После этого вы увидите фактическое сообщение SOAP:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <FindUsersFollowersResponse xmlns="http://tempuri.org/">
```

```

<FindUsersFollowersResult
xmlns:a="http://schemas.datacontract.org/2004/07/
AccountManagement" xmlns:i="http://www.w3.org/2001/
XMLSchema-instance">
  <a:Follower>
    <a:FollowerId>follower_0</a:FollowerId>
    <a:FollowerName>happy follower 0</a:FollowerName>
    <a:SocialTags
xmlns:b="http://schemas.microsoft.com/2003/10/
Serialization/Arrays">
      <b:string>programming</b:string>
      <b:string>DDD</b:string>
      <b:string>Psychology</b:string>
    </a:SocialTags>
  </a:Follower>
<a:Follower>
...

```

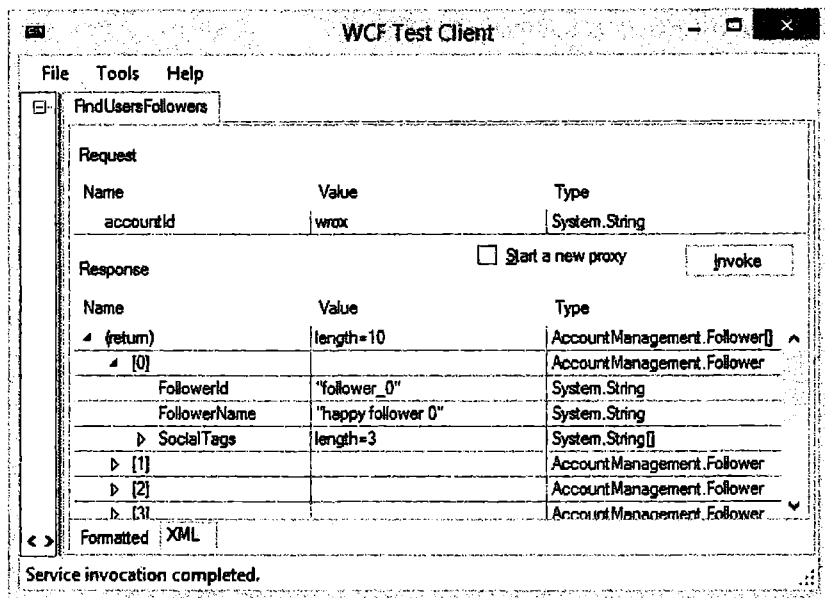


Рис. 13.4. RPC-вызов службы WCF в тестовом клиенте

Создание клиентов службы WCF

Теперь пришло время узнать, какой объем рутинной работы способна выполнить за вас комбинация WCF, SOAP и Visual Studio. Сначала создайте проект ограниченного контекста поиска, а затем вы увидите, как выполнять вызовы RPC ограниченного контекста управления учетными записями, просто передавая ограниченному контексту поиска URL-адрес ограниченного контекста управления учетными записями.

Для начала нужно добавить в решение новый проект типа WCF Service Application с именем *Discovery*, представляющий ограниченный контекст поиска. Затем внутри этого ограниченного контекста нужно добавить службу WCF с именем *Recommender*. Она будет представлять веб-службы, которые веб-сайт сможет использовать для получения списка пользователей-единомышленников. Чтобы освободить память, взгляните еще раз на схему, изображенную на рис. 13.1.

Служба *Recommender* несет двойную ответственность. Во-первых, она предоставляет API для клиентов, запрашивающих список возможных единомышленников. Для выполнения этой обязанности она производит RPC-вызовы к ограниченному контексту управления учетными записями, чтобы получить учетные записи единомышленников. Чтобы создать реализацию этой службы, нужно запустить проект *AccountManagement*. (Выберите его в панели *Solution Explorer* (Обозреватель решения) и нажмите **Ctrl+F5**.) Затем убедитесь, что он запустился, напрямую обратившись к нему из веб-браузера по адресу *http://localhost:3100/FollowerDirectory.svc*. Если откроется страница с заголовком «FollowerDirectory Service», значит, все в порядке.

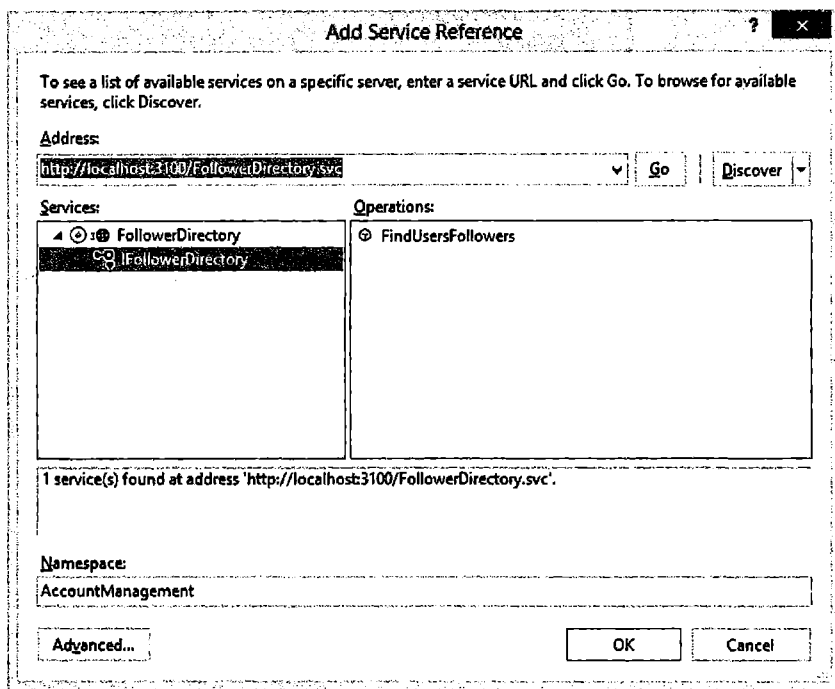


Рис. 13.5. Добавление ссылки на службу в Visual Studio

Далее нужно передать URL в Visual Studio, чтобы сгенерировать прокси-классы. Если щелкнуть правой кнопкой мыши на узле *References* в проекте *Discovery* в панели *Solution Explorer* (Обозреватель решения) и выбрать в контекстном меню пункт *Add Service Reference* (Добавить ссылку на службу), откроется диалог, где можно

вставить URL в поле **Address** (Адрес). После этого останется только ввести в поле **Namespace** (Пространство имен), в нижней части диалога **Add Service Reference** (Добавить ссылку на службу), значение **AccountManagement** и щелкнуть на кнопке **Go** (Вперед). Окно диалога должно напоминать изображение на рис. 13.5, где показан распахнутый узел **FollowerDirectory**, соответствующий обнаруженной и идентифицированной веб-службе. Теперь можно щелкнуть на кнопке **OK**.

Чтобы увидеть сгенерированные прокси-классы, распахните узел **Discovery**. **AccountManagement**, который был добавлен в папку **Service References** (Ссылки на службы). На рис. 13.6 показаны сгенерированные прокси-классы. Эти классы вы вскоре будете использовать для создания объектов в ограниченном контексте поиска и вызова методов ограниченного контекста управления учетными записями с помощью механизма **RPC** (через **SOAP/HTTP**).

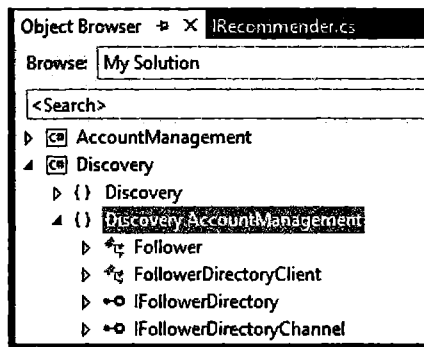


Рис. 13.6. Сгенерированные прокси-классы

Последний шаг — создание веб-службы **Recommender**, которая включит сгенерированные прокси-классы в работу. В листингах 13.4 и 13.5 показаны простейшие реализации **IRecommender** и **Recommender**, достаточные для демонстрации вызовов **RPC**. Добавьте этот программный код в проект **Discovery**.

Листинг 13.4. Интерфейс **IRecommender**, определяющий веб-службу

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace Discovery
{
    [ServiceContract]
    public interface IRecommender
    {
        [OperationContract]
        List<string> GetRecommendedUsers(string accountId);
    }
}
```


Листинг 13.5. Класс *Recommender*, реализующий веб-службу и выполняющий вызовы RPC

```
using System;
using System.Linq;
using System.Collections.Generic;
using Discovery.AccountManagement;

namespace Discovery
{
    public class Recommender : IRecommender
    {
        public List<string> GetRecommendedUsers(string accountId)
        {
            // создать экземпляр прокси-класса
            var accountManagementBC =
                new AccountManagement.FollowerDirectoryClient();

            // вызвать метод прокси-класса – он иницирует RPC-вызов через
            // SOAP/HTTP
            var followers = accountManagementBC.FindUsersFollowers(accountId);
            return FindRecommendedUsersBasedOnSocialTags(followers);
        }

        private List<string> FindRecommendedUsersBasedOnSocialTags(
            Follower[] followers)
        {
            /*
             * Действующая система могла бы просматривать теги пользователей и
             * искать популярные учетные записи с похожими тегами, производя
             * большее число RPC-вызовов.
             */
            var tags = followers.SelectMany(f => f.SocialTags).Distinct();
            return tags.Select(t => t + "_user_1").ToList();
        }
    }
}
```

В листинге 13.5 создается экземпляр *AccountManagement.FollowerDirectoryClient*. Это прокси-класс, сгенерированный Visual Studio при добавлении ссылки на службу. Когда вызывается его метод *FindUsersFollowers()*, он запускает RPC-вызов по сети и вызывает метод, добавленный вами в ограниченный контекст управления учетными записями. Самое главное здесь состоит в том, что WCF и Visual Studio позаботились о фактической реализации сетевых взаимодействий. Большая часть кода, который вы добавите, будет выглядеть аналогично, даже если он не будет связан с сетевыми взаимодействиями.

Вы можете проверить работу всего комплекса, настроив запуск обоих проектов (как было показано в предыдущей главе — щелчком правой кнопки на решении в панели *Solution Explorer* (Обозреватель решения) и настройкой свойства *Startup Project* (Запуск проекта)) и нажав F5, предварительно выбрав *Recommender.svc* в панели *Solution Explorer* (Обозреватель решения). На экране появится окно тестового

клиента WCF, в котором вам нужно попробовать вызвать `GetRecommendedUsers()`, как показано на рис. 13.7.

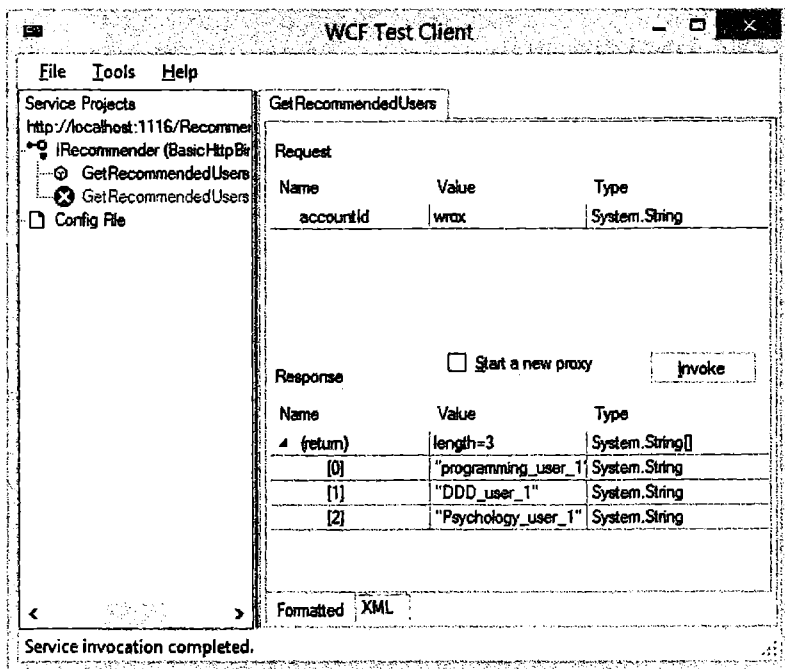


Рис. 13.7. Должен быть выполнен RPC-вызов

В результате RPC-вызова (рис. 13.7) будут получены данные, «зашифрованные» в программный код веб-службы. Этот эксперимент показывает, что RPC-вызов между двумя ограниченными контекстами был успешно выполнен. Определенно, цель данного примера — интеграция ограниченных контекстов управления учетными записями и поиска без образования зависимости на уровне двоичного кода — достигнута.

Недостатки SOAP

Несмотря на существование большого количества общедоступных SOAP API, новые API на основе SOAP больше не создаются. Одной из самых больших проблем SOAP является сложность и излишняя подробность формата сообщений, который вы могли видеть выше. Люди критически относятся к ненужной сложности, и они часто указывают на SOAP как на яркий пример такой ненужной сложности. Поэтому не следует бездумно экспортировать общедоступные SOAP API, но это также не значит, что нужно осторожничать, выбирая SOAP для внутренних нужд.

В настоящее время при реализации RPC через HTTP предпочтение отдается легковесным форматам на основе простой разметки XML или JSON. Следующий раздел демонстрирует примеры такого подхода с использованием ASP.NET Web API.

Простой формат XML или JSON: современный подход к реализации RPC

Чтобы узнать, как осуществить интеграцию через HTTP без сложностей, свойственных формату SOAP, в этом разделе вы повторно реализуете пример интеграции ограниченных контекстов социальной сети, но на этот раз с использованием легковесного формата JSON. Ниже приводится JSON-версия сообщения SOAP, показанного выше. Вы можете вернуться назад и сравнить эти два формата, чтобы по достоинству оценить компактность JSON.

```
{
  "followers": [
    {
      "accountId": "34djdlfjk2j2",
      "socialTags": [
        "ddd", "soa", "tdd", "kanban"
      ]
    },
    ...
  ]
}
```

ПРИМЕЧАНИЕ

Поскольку этот пример является повторной реализацией предыдущего примера SOAP/WCF, схема, изображенная на рис. 13.1, также применима здесь. При желании вы можете вернуться к этому рисунку, чтобы освежить память.

Реализация RPC через HTTP с использованием формата JSON и ASP.NET Web API

ASP.NET Web API — это новейший фреймворк компании Microsoft, предназначенный для создания веб-служб. Ниже вы узнаете, что он является отличным выбором для построения RESTful API. А пока посмотрим, как с его помощью создавать JSON RPC API. Для начала создайте в Visual Studio новое решение с именем `PPPPDD.Json.SocialMedia`. Добавьте в это решение проект типа ASP.NET Web Application с именем `AccountManagement`. При создании этого проекта выберите шаблон `Empty` (Пустой) и установите флажок `Web API`. После создания настройте проект, чтобы он всегда использовал порт 3200.

Код, который должен выполняться для обработки веб-запросов к вашему Web API, находится в контроллерах. Контроллеры можно рассматривать как средство выражения предметных понятий на едином языке, но старайтесь не включать в них предметную логику. Некоторые разработчики рассматривают контроллеры как прикладные службы.

ПРИМЕЧАНИЕ

Прикладные службы (Application Services) — уровень служб подробно рассматриваются в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования».

Для начала добавьте класс с именем `FollowerDirectoryController` в папку `Controllers`, находящуюся в корне проекта; приложения типа Web API требуют, чтобы контроллеры находились в этой папке. Реализация класса `FollowerDirectoryController` приводится в листинге 13.6.

Листинг 13.6. `FollowerDirectoryController`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace AccountManagement.Controllers
{
    public class FollowerDirectoryController : ApiController
    {
        public IHttpActionResult GetUsersFollowers(string accountId)
        {
            var followers = GenerateDummyFollowers().ToList();
            return Json(followers);
        }

        private IEnumerable<Follower> GenerateDummyFollowers()
        {
            for (int i = 0; i < 10; i++)
            {
                yield return new Follower
                {
                    FollowerId = "follower_" + i,
                    FollowerName = "happy follower " + i,
                    SocialTags = new List<string>
                    {
                        "programming", "DDD", "Psychology"
                    }
                };
            }
        }
    }

    public class Follower
    {
        public string FollowerId { get; set; }

        public string FollowerName { get; set; }

        public List<string> SocialTags { get; set; }
    }
}
```

Для целей демонстрации класс `FollowerDirectoryController` в листинге 13.6 возвращает жестко заданный список `Followers` (в формате JSON). В действующем приложении этот класс, скорее всего, будет выполнять поиск в базе данных или вызывать другие методы для получения информации о возможных единомышленниках.

Запустив приложение (нажатием клавиши F5), вы сможете проверить новую веб-службу, обратившись к ней из браузера. В соответствии с соглашениями по умолчанию, действующими для проектов типа Web API, служба доступна по адресу `http://localhost:3200/api/followerdirectory/getusersfollowers?accountId=123`, где значение для параметра `accountId` является переменной. (В данном примере это значение может быть чем угодно.) На рис. 13.8 показан результат обращения к API из браузера, в окне которого можно видеть полученный ответ в формате JSON.

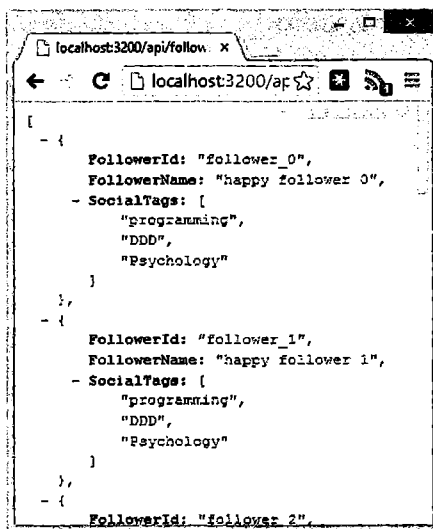


Рис. 13.8. Ответ контроллера Web API в окне браузера

ПРИМЕЧАНИЕ

Полезно будет установить расширение, форматирующее сообщения в формате JSON при отображении в браузере. Для браузера Chrome такое расширение называется JSONView (<https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en>). Аналогичные расширения существуют также для браузеров Firefox, Internet Explorer и др.

Как вы могли заметить, этот подход требует приложить чуть больше усилий по настройке контроллера в сравнении с подходом на основе SOAP и WCF, но данные, передаваемые по сети, намного чище и легче и в них легко разобраться, если возникнет такая потребность. Это дополнительное преимущество пригодится во время отладки.

В отсутствие богатства метаданных, предоставляемых SOAP, нет никакой возможности автоматически сгенерировать прокси-классы для простого JSON API. Однако это не является такой уж большой проблемой, в чем вы сейчас убедитесь самостоятельно.

Чтобы создать клиент JSON API, добавьте в решение новый проект типа ASP.NET Web Application с именем *Discovery*, представляющий ограниченный контекст поиска. Внутри проекта *Discovery* добавьте в папку *Controllers* класс *RecommenderController*. Реализация этого класса приводится в листинге 13.7. Для его компиляции нужно установить пакеты *HttpClient* и *ServiceStack.Text*, выполнив следующие команды в консоли диспетчера пакетов Nuget:

```
Install-Package Microsoft.AspNet.WebApi.Client -Project Discovery
Install-Package ServiceStack.Text -Project Discovery
```

Листинг 13.7. *RecommenderController*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using ServiceStack.Text;

namespace Discovery.Controllers
{
    public class RecommenderController : ApiController
    {
        public List<string> GetRecommendedUsers(string accountId)
        {
            var accountManagementUrl =
                "http://localhost:3200/api/" +
                "followerdirectory/getusersfollowers?" +
                "accountId=" + accountId;

            var response = new WebClient().DownloadString(accountManagementUrl);
            var followers = JsonSerializer
                .DeserializeFromString<List<Follower>>(response);
            // автоматически преобразуется в JSON службой Web API
            return FindRecommendedUsersBasedOnSocialTags(followers);
        }

        private List<string> FindRecommendedUsersBasedOnSocialTags(
            List<Follower> followers)
        {
            /*
             * Действующая система могла бы просматривать теги пользователей и
             * искать популярные учетные записи с похожими тегами, производя
             * большее число RPC-вызовов.
             */
            var tags = followers.SelectMany(f => f.SocialTags).Distinct();
            return tags.Select(t => t + "_user_1").ToList();
        }
    }
}
```

```
}

/* этот класс не является общим для ограниченных контекстов,
 * в целях избежать зависимости на уровне исходных текстов
 */
public class Follower
{
    public string FollowerId { get; set; }

    public string FollowerName { get; set; }

    public List<string> SocialTags { get; set; }
}
}
```

Как можно заметить в листинге 13.7, логика реализации в нем напоминает подход на основе WCF, представленный в предыдущем примере. Однако в этом решении потребовалось больше ручного труда для отправки HTTP-запросов и парсинга ответов. Впрочем, как показывает пример, компания Microsoft и сообщество создали множество библиотек с богатыми наборами функций, которые берут на себя массу рутинной работы.

ВНИМАНИЕ

В листинге 13.7 представлен чрезмерно упрощенный подход к интеграции. В действующем решении может понадобиться использовать асинхронные клиенты и выполнять проверку ошибок. Также часто бывает желательно устанавливать HTTP-заголовки, такие как Ассерт, чтобы указать наиболее предпочтительный формат (если удаленная служба поддерживает несколько форматов, таких как XML, JSON, HTML и CSV).

Чтобы убедиться, что все работает, как задумывалось, нужно настроить запуск обоих проектов (как было показано в предыдущих примерах). Затем открыть URL только что созданного API `GetRecommendedUsers`. Этот URL имеет вид: `http://localhost:{port}/api/recommender/getrecommendedusers?accountId=123`, где часть «port» должна соответствовать номеру порта, который прослушивает ограниченный контекст поиска на вашем компьютере. Если нажать клавишу F5 в решении, автоматически запустится браузер, где вы сможете увидеть номер порта (номер порта можно также указать вручную, как было показано выше в этой главе).

Выбор варианта RPC

Только что была продемонстрирована возможность интеграции ограниченных контекстов с RPC посредством HTTP с использованием двух основных подходов. С помощью WCF и SOAP достаточно просто добавить несколько атрибутов в предметную модель и внезапно получить распределенную систему, свободную от любых зависимостей между ограниченными контекстами на уровне двоичного кода. Это, в свою очередь, обеспечит большую независимость группам разработчиков и позволит им не волноваться, что какие-то их решения нарушат работоспособность других ограниченных контекстов. Однако одной из проблем SOAP является слож-

ный и излишне подробный формат сообщений; во многих случаях механизмы RPC используются для интеграции в самых простых случаях, поэтому такие сложности выглядят нелогичными. Именно по этой причине в настоящее время большей популярностью пользуется подход на основе простого формата XML или JSON.

Оба варианта, однако, страдают врожденными недостатками RPC, которые упоминались в главе 11. Прежде всего, они труднее поддаются масштабированию. Взгляните еще раз на схему, изображенную на рис. 13.1, и представьте, что владельцы предприятия потребовали увеличить скорость поиска возможных единомышленников и их отображения на экране. Чтобы увеличить общую производительность, может потребоваться масштабировать оба ограниченных контекста — поиска и управления учетными записями. Если цепочка вызовов RPC будет охватывать три ограниченных контекста, тогда может потребоваться масштабировать три ограниченных контекста из-за временной зависимости между ними.

Если говорить об отказоустойчивости, здесь также есть некоторые причины для беспокойства. Если ограниченный контекст управления учетными записями откажет, ограниченный контекст поиска тоже перестанет работать, потому что не сможет выполнить запрос RPC и получить список возможных единомышленников. И снова проблема объясняется временной зависимостью.

Может показаться, что вы стоите перед выбором: либо слабосвязанная платформа и интеграция посредством HTTP, либо масштабируемая и отказоустойчивая система, но с использованием фреймворка обмена сообщениями, такого как NServiceBus. В действительности это совершенно не так. Следующий раздел в этой главе показывает, что есть возможность получить масштабируемость и отказоустойчивость систем обмена сообщениями и слабосвязанную платформу HTTP, соединив принципы реактивного программирования с архитектурой REST.

REST

В этом разделе вы в третий раз займетесь интеграцией ограниченных контекстов приложения социальной сети. Но на этот раз реализация будет полностью переработана для достижения максимальной масштабируемости, отказоустойчивости и эффективности разработки. Эта третья версия все еще опирается исключительно на применение протокола HTTP вместо тяжеловесного фреймворка обмена сообщениями. Однако здесь будут применены принципы реактивного программирования, SOA и ослабления зависимостей, представленные в главе 11.

Термин REST часто неправильно понимают и неправильно употребляют. Поэтому, прежде чем приступить к созданию приложения на основе архитектуры RESTful, очень важно, чтобы вы поняли, что в действительности означает термин REST.

Разоблачение мифов REST

Термин REST был введен Роем Филдингом (Roy Fielding). Он создал REST как архитектурный стиль, опирающийся на принципы, сделавшие Интернет столь успешным. REST включает множество основополагающих понятий, в том числе

ресурсы и гиперсреду (hypermedia), которые служат платформой для расширяемых клиентов и серверов.

ПРИМЕЧАНИЕ

Рой Филдинг ввел в обиход термин REST в своей докторской диссертации «Architectural Styles and the Design of Network-Based Software Architectures». Текст диссертации (на английском языке) доступен по адресу <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

Ресурсы

Запросы HTTP в системе RESTful адресуются ресурсам. Ответы содержат запрашиваемые ресурсы (в случае успеха). Ресурсами могут быть документы, такие как веб-страницы, или мультимедийные файлы, такие как файлы MP3. Ресурсы отлично согласуются с философией DDD, потому что понятия предметной области могут выражаться в терминах ресурсов — дальнейшего расширения единого языка. Как простой пример рассмотрим финансовую предметную область, где могут выполняться транзакции, перечисляющие средства со счета на счет. В едином языке могли бы иметься отдельные термины для каждого типа транзакций, такие как «B2B Transaction» (перечисления между предприятиями) или «Personal Transaction» (расчеты с частным лицом). Эти транзакции могли бы экспортироваться как ресурсы, доступные по унифицированным идентификаторам ресурсов (Uniform Resource Identifier, URI, или URL) <http://pppdddemo.com/B2bTransactions> или <http://pppdddemo.com/PersonalTransactions>. Это существенное отличие от RPC, где запросы и ответы интерпретируются как вызовы методов с использованием императивных имен¹.

ВНИМАНИЕ

Между унифицированными идентификаторами ресурсов (URI) и унифицированными указателями ресурсов (URL) есть одно важное отличие, о котором следует знать. Их не следует воспринимать как одно и то же. В действительности унифицированные указатели ресурсов (URL) — это лишь одна из разновидностей унифицированных идентификаторов ресурсов (URI); кроме нее существуют другие, менее распространенные разновидности. Узнать больше о различиях можно в Википедии (<https://ru.wikipedia.org/wiki/URI>).

Ресурсы связаны с представлениями отношением «один ко многим». Иными словами, запрашивая ресурс, можно указать другой протокол или тип содержимого, такой как JSON, XML или HTML. Все ответы будут содержать тот же самый ресурс, но в разных представлениях, определяемых синтаксическими правилами запрошенного формата. Чуть ниже будет показано, как клиенты RESTful API могут выбирать формат, указывая в HTTP-заголовке Accept тип, как определяется многоцелевыми расширениями почты в Интернете (Multipurpose Internet Mail Extensions, MIME).

¹ Имен, похожих на команды или инструкции. — *Примеч. пер.*

Ниже приводится еще пара важных деталей о ресурсах.

- Между унифицированными идентификаторами ресурсов (URI) и ресурсами существует отношение «многие к одному». То есть одному ресурсу может соответствовать несколько URI.
- Ресурсы могут быть иерархическими. Например, экспортировать адрес пользователя можно было бы с помощью URI, такого как `/accounts/user123/address`, где каждый сегмент пути является дочерним ресурсом сегмента, стоящего слева, в точности как папки в пути к файлу.

Гиперсреда

Люди, путешествуя во Всемирной паутине, переходят от одной веб-страницы к другой, щелкая по ссылкам. Это — гиперсреда в действии. При наличии гиперссылок в ресурсах компьютеры тоже получают возможность переходить от ресурса к ресурсу, просто следуя по ссылкам. Этот прием будет показан далее в данной главе.

Гиперсреда дает практикам DDD еще одну возможность более явно выразить предметную область. Представьте процедуру страхования автомобиля. Все шаги прикладного процесса можно было бы представить в виде ссылок в гиперсреде на следующие возможные шаги, выраженные с использованием единого языка. Мало того что такой подход помогает выразить предметные понятия, его также можно использовать для моделирования рабочих процедур или предметных процессов.

Использование гиперсреды для организации межмашинных взаимодействий означает, что клиенты RESTful API не связаны с его URI. Это обеспечивает отсутствие тесной связи между клиентами и серверами и их свободное и независимое развитие. Одна из основных причин, почему многие отдают предпочтение архитектуре REST, заключается в том, что решения на основе SOAP обычно получаются хрупкими из-за образования тесных связей между серверами и клиентами.

Отсутствие информации о состоянии

Состояние приложения в архитектуре RESTful, например товары в корзине покупателя, не должно храниться на стороне сервера. Это открывает дополнительные возможности для улучшения отказоустойчивости и масштабируемости, потому что клиенты не должны вновь и вновь обращаться к одной и той же машине, где хранится информация о состоянии. Соответственно информация о состоянии должна храниться на стороне клиента и посылаться на сервер с каждым запросом. Вернемся к примеру с корзиной покупателя. В REST API без сохранения состояния товары, находящиеся в корзине, могли бы храниться в виде блоков данных cookies и пересылаться на сервер с каждым запросом. Благодаря этому никакие проблемы на сервере не смогут воспрепятствовать обработке таких запросов другими серверами.

ПРИМЕЧАНИЕ

На практике отсутствие информации о состоянии на стороне сервера может повлечь необходимость идти на различные компромиссы, и вам, возможно, придется принимать решения исходя из прагматичных соображений. Однако стремление отказаться от хранения информации о состоянии везде, где это возможно, является достойной внимания философией.

REST в полной мере соответствует протоколу

Протокол HTTP поддерживает ряд соглашений, обеспечивающих основу масштабируемости, отказоустойчивости и отсутствие тесных связей. Так как REST основывается на тех же принципах, которые обеспечили успех Всемирной паутины, очень важно, чтобы вы имели хотя бы общее представление об особенностях http, прежде чем приступите к созданию приложений с архитектурой RESTful.

Глаголы

Протокол HTTP предоставляет универсальный интерфейс для взаимодействий с ресурсами. Например, чтобы получить ресурс, нужно послать запрос GET с URI этого ресурса. Чтобы удалить ресурс, нужно послать запрос DELETE с URI удаляемого ресурса. Чтобы создать ресурс с желаемым идентификатором URI, можно послать запрос PUT. Чтобы добавить элементы в коллекцию, можно воспользоваться глаголом¹ POST. Примеры типичных глаголов, используемых для работы с ресурсами, приводятся в табл. 13.1.

Таблица 13.1. Использование глаголов HTTP для создания, чтения, изменения и удаления ресурсов

URI	Глагол	Действие
/accounts/user123	GET	Чтение/извлечение ресурса
/accounts/user123	DELETE	Удаление ресурса
/accounts/user123	PUT	Создание ресурса
/accounts/user123/addresses	POST	Изменение ресурса

ВНИМАНИЕ

Таблица 13.1 не является строгим определением семантики глаголов HTTP. Самое примечательное, что до сих пор не утихают споры о том, когда использовать методы PUT и POST. Дополнительную информацию по данной теме можно найти в книге «The RESTful Cookbook» (<http://restcookbook.com/HTTP%20Methods/put-vs-post/>)².

Имея единое множество глаголов, которые повсюду в Интернете применяются единообразно, легко создать обобщенный прикладной интерфейс для клиентов и компоненты инфраструктуры, такие как кэши, соответствующие соглашениям, принятым в Веб. Подумайте сами — для каждого языка программирования имеются библиотеки поддержки HTTP, которые вы можете использовать в сочетании с мощью общепринятых соглашений для интеграции ограниченных контекстов.

¹ Здесь имеется в виду, что названия HTTP-запросов имеют форму глаголов-команд, например: GET (получить), POST (послать), DELETE (удалить) и т. д. — *Примеч. пер.*

² Описание семантики глаголов HTTP на русском языке можно найти по адресу <http://www.restapitutorial.ru/lessons/httpmethods.html>. — *Примеч. пер.*

Коды состояния

Глаголы HTTP дополняются их кодами состояния. По аналогии с глаголами, наличие единого множества кодов состояния означает, что для любого веб-агента будут действовать одни и те же соглашения. Например, при попытке послать запрос к несуществующему URI вы получите код состояния HTTP 404, потому что это общий стандарт для всех систем, придерживающихся его.

Коды состояния HTTP группируются по первой цифре, как показано в табл. 13.2. Внутри каждой группы определены более конкретные коды.

Таблица 13.2. Группы кодов состояния HTTP

Группа	Определение	Пример
1xx	Информационные	Эти коды редко используются на практике
2xx	Успех	Запрошенный ресурс возвращен
3xx	Перенаправление	Запрошенный ресурс перемещен по другому адресу
4xx	Ошибка клиента	Клиентом указано ошибочное значение параметра
5xx	Ошибка сервера	В программном коде API произошла ошибка, не позволявшая вернуть ресурс

Желающие узнать больше смогут найти в Википедии доступное введение в коды состояния HTTP (https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP).

Заголовки

Возможно, вы знаете, что кроме URI и тела запросы/ответы HTTP имеют также заголовки, в которых передается дополнительная информация. Системы RESTful часто используют заголовки HTTP, управляющие кэшированием, о которых рассказывается далее в этой главе.

Большинство приложений RESTful требуют обеспечения некоторого уровня безопасности. Так как REST по своей природе не имеет состояния, часто рекомендуется передавать в заголовках информацию для аутентификации и авторизации, используя такие протоколы, как OAuth.

Дополнительную информацию о заголовках, доступных в запросах и ответах HTTP, можно найти в достаточно подробной статье в Википедии (https://ru.wikipedia.org/wiki/Список_заголовков_HTTP).

ПРИМЕЧАНИЕ

Обсуждение вопросов аутентификации и авторизации в системах RESTful выходит далеко за рамки этой книги. Замечательным бесплатным ресурсом для изучения этой темы является электронная книга «The RESTful CookBook» (<http://restcookbook.com/Basics/loggingin/>).

Чем не является REST

Архитектура REST может быть отличным выбором для одних проектов и недостаточно хорошим для других. Какой бы выбор вы ни сделали, всегда важно называть вещи своими именами, чтобы никого не вводить в заблуждение. К сожалению, термин REST часто используют неправильно. Поэтому, прежде чем назвать свой прикладной интерфейс «RESTful API», убедитесь, что он как минимум основывается на понятиях ресурсов и гиперсреды.

После многочисленных резонансных случаев неправильного употребления термина REST в 2008 году Рой Филдинг был вынужден написать в блоге статью, где описывались минимальные требования к API, которые могут называться RESTful (<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>)¹. Другой контрмерой против неправильного использования термина REST является модель зрелости Ричардсона (Richardson Maturity Model, <http://martinfowler.com/articles/richardsonMaturityModel.html>). Эта модель может действовать как барометр, указывающий, насколько близок ваш API к определению RESTful.

REST для интеграции ограниченных контекстов

Оставшаяся часть данной главы будет посвящена повторной реализации примеров интеграции ограниченных контекстов, которые обсуждались выше. Благодаря появлению новых возможностей и агрессивной маркетинговой кампании, число регистраций пользователей в зарождающихся фиктивных социальных сетях снова увеличивается в геометрической прогрессии. Бизнес стремится извлечь максимум выгоды из этого успеха, предлагая премиум-аккаунты постоянным пользователям. Премиум-аккаунты — это один из способов привлечения последователей на веб-сайты социальных сетей и, соответственно, дополнительных прибылей от расширения круга пользователей.

К сожалению, как это часто бывает, интеграция на основе RPC плохо поддается масштабированию. Выбор этой технологии вполне оправдан на начальном этапе, когда важно максимально сократить период от замысла до воплощения, но теперь она мешает дальнейшему развитию, потому что разработчики слишком много времени тратят на «борьбу с пожаром». Поэтому, прежде чем добавлять новые особенности, нужно стабилизировать систему, обеспечив поддержку быстрого роста предприятия.

Эта новая версия системы основана на архитектуре, управляемой событиями. Примечательно, что для устранения препятствий, мешающих развитию системы, вместо шины сообщений (распространенного подхода, представленного в предыдущей главе) эта система использует REST и HTTP.

¹ Некоторые требования перечислены в статье Википедии: <https://ru.wikipedia.org/wiki/REST>. — *Примеч. пер.*

Подготовка к переходу на использование REST

Как отмечалось в предыдущей главе, этап начального проектирования поможет получить общее представление и более глубокое понимание, как создаваемая система будет реализовывать функциональные и нефункциональные требования к ней. Некоторые шаги в процессе проектирования системы, использующей архитектуру REST для интеграции, будут отличаться от шагов, использовавшихся при проектировании системы обмена сообщениями. Однако большинство шагов остались прежними, включая первый: описание предметной области.

DDD

Как и прежде, полезным первым шагом в процессе проектирования системы является выражение деловой логики с использованием единого языка в виде множества схем. Практически всегда есть смысл определить, какую задачу требуется решить и какие предметные процессы смоделировать, прежде чем приступить к выбору технических решений.

На рис. 13.9 изображена диаграмма компонентов, иллюстрирующая новую, управляемую событиями архитектуру сценария использования «Рекомендация единомышленников». Так же как в решении на основе обмена сообщениями из предыдущей главы, здесь в центре внимания находятся предметные команды и события. Фактически эту архитектуру можно было бы использовать для реализации системы обмена сообщениями, потому что диаграмма отражает потоки сообщений в сценарии использования и не зависит от выбора технологий.

На рис. 13.9 можно видеть два основных предметных события. Первое: «Следовать за единомышленником» (Began Following). Все сотрудники компании понимают, что предметное событие «Следовать за единомышленником» возникает, когда один пользователь начинает следовать за другим. Это часть единого языка и одно из основных предметных понятий. Второе предметное событие: «Рекомендованные премиум-аккаунты определены» (Premium Recommendations Identified). Это тоже часть единого языка, представляющая события обнаружения ограниченным контекстом поиска премиум-аккаунтов, которые можно рекомендовать обычному пользователю для следования. Это важнейшее предметное понятие, потому что популяризация премиум-аккаунтов среди обычных пользователей является одной из главных особенностей новой бизнес-модели.

SOA

В главе 11 было показано, что следование принципам SOA помогает создавать слабосвязанные ограниченные контексты. Это обстоятельство, в свою очередь, может служить основой высокой производительности групп разработчиков. Принципы SOA не зависят от выбора технологий, поэтому их можно применять при создании систем, использующих HTTP.

Благодаря изоляции ограниченных контекстов, разрабатываемых отдельными группами, достигается их слабая связанность. Каждая группа может разрабатывать свои особенности в соответствии со своими приоритетами, не отвлекаясь на решение проблем с зависимостями. Единственное отличие решения на основе

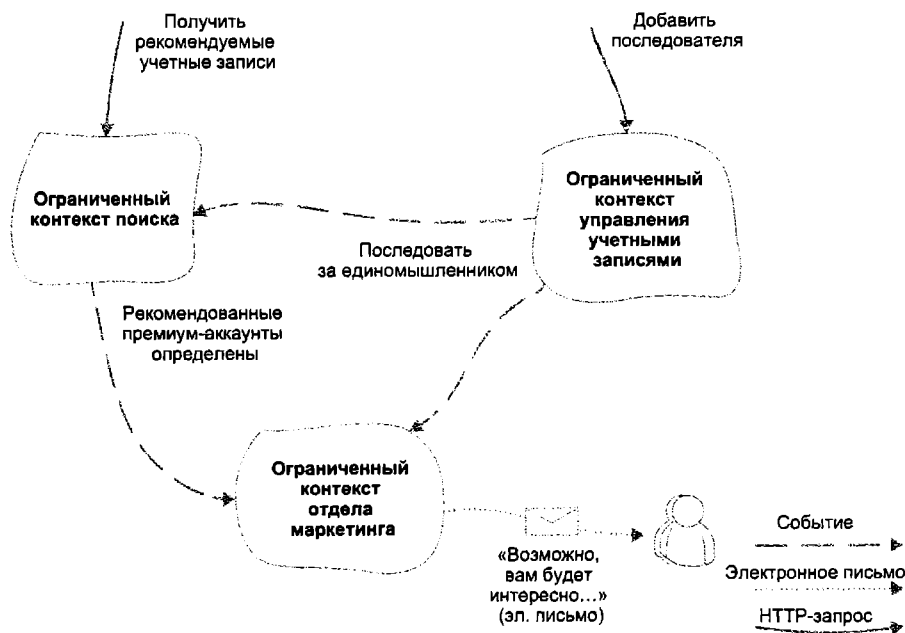


Рис. 13.9. Диаграмма компонентов для сценария использования «Рекомендация единомышленников»

SOA и HTTP, в сравнении с обменом сообщениями, — контракт между группами — заключается не в виде классов в программном коде, а в виде форматов запросов и ответов HTTP. Это в полной мере демонстрируют примеры, следующие далее.

Событийное и реактивное программирование

В главах 11 и 12 вы узнали, что для создания масштабируемых и отказоустойчивых систем рекомендуется использовать асинхронный обмен сообщениями, основанный на принципах реактивного программирования. То же относится к интеграции посредством REST и HTTP. Как вы уже, возможно, знаете, протокол HTTP по своей природе не поддерживает шаблон публикация/подписка, поэтому нет никакой возможности публиковать события для подписчиков, как в случае с шиной сообщений. Вместо этого при использовании архитектуры REST клиенты периодически запрашивают информацию о происшедших изменениях. Необходимость выполнения таких запросов обычно отрицательно сказывается на масштабировании, но использование соглашений о кэшировании в HTTP может ослабить эту проблему.

ПРИМЕЧАНИЕ

Технически можно создать масштабируемую, управляемую событиями систему на основе REST без использования запросов информации об изменениях. Каждый источник событий может хранить список подписчиков и соответствующие адреса URL для передачи сообщений о публикации события. Однако этот подход обычно сложнее в реализации и часто вступает в противоречие с принципом, требующим не хранить информацию о состоянии на стороне сервера.

На рис. 13.10 изображена диаграмма контейнеров для новой реактивной системы социальной сети на основе RESTful, которую мы будем создавать в оставшейся части этой главы. Обратите внимание на некоторое сходство с системой, основанной на обмене сообщениями: каждый компонент настолько мал, что может масштабироваться независимо и в соответствии с потребностями предприятия; ограниченные контексты не имеют общих зависимостей, таких как базы данных. Кроме того, благодаря применению протокола HTTP каждая группа может выбирать любые технологии для реализации своего ограниченного контекста (чего труднее было достичь в предыдущей главе). Эти особенности обеспечивают поддержку масштабируемости, отказоустойчивости и высокой скорости разработки.

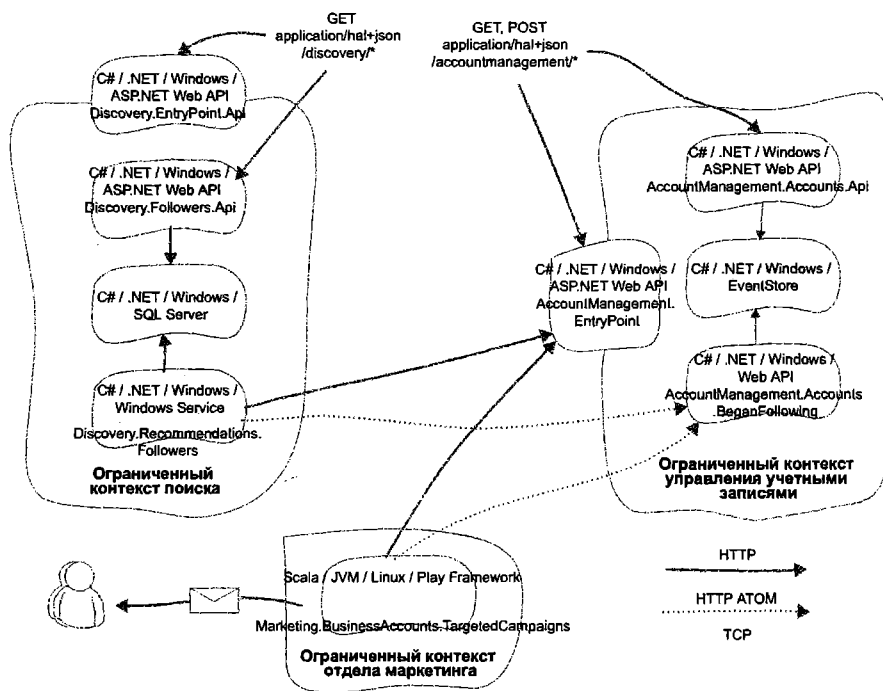


Рис. 13.10. Диаграмма контейнеров ограниченных контекстов поиска, управления учетными записями и маркетинга

Большое значение для масштабируемости имеет степень дробления проектов. Для реализации HTTP API можно было бы поместить все конечные точки в один проект, но это лишит вас возможности разворачивать их независимо друг от друга, исходя из потребностей в масштабировании. В этой главе рекомендуется создавать отдельный проект для каждого ресурса. Примерами такой организации на рис. 13.10 могут служить отдельные проекты для ресурса точки входа и ресурса Accounts.

Иногда, при наличии вложенных ресурсов, бывает желательно выделить их в отдельные проекты. Главным фактором в таких решениях обычно является компо-

мисс между сложностью сопровождения дополнительных проектов и возможностью независимого масштабирования API. У вас может даже появиться желание переместить отдельные обработчики запросов в собственные проекты, если конкретные сценарии использования имеют особые требования к масштабируемости.

HTTP и гиперсреда

Гиперсреда является основой REST, потому что позволяет развивать клиентские и серверные компоненты независимо друг от друга. Поэтому, прежде чем приступать к созданию системы, полезно немного подумать о контрактах взаимодействий в гиперсреде. (Вы все еще можете использовать итеративный подход по мере продвижения.) Отличный способ проектирования процессов в REST заключается в создании диаграмм последовательностей операций, одна из которых изображена на рис. 13.11 и демонстрирует управляемый событиями сценарий использования «Добавление последователя».

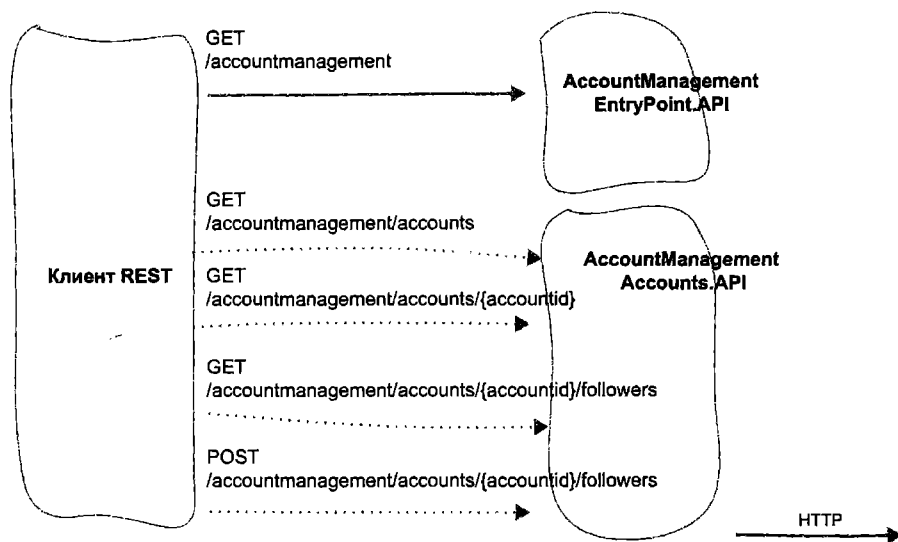


Рис. 13.11. Последовательность HTTP-запросов для сценария использования «Добавление последователя»

Как видите, клиенты связаны только с URI точки входа. Используя его, они выполняют запросы к другим службам, следуя по ссылкам в гиперсреде, возвращаемым в ответах. Например, клиентам, инициирующим сценарий использования «Добавление последователя» (аналог предметной команды) в ограниченном контексте управления учетными записями, нужен всего лишь URI ресурса точки входа — `/accountmanagement`. Далее клиенты просто следуют по ссылкам в гиперсреде, возвращаемым в HTTP-ответах, пока не достигнут ресурса `Followers`.

На диаграмме контейнеров (см. рис. 13.10) рядом с каждой конечной точкой HTTP для справки указан тип ее содержимого. В большинстве случаев это

application/hal+json. Аббревиатура HAL расшифровывается как Hypertext Application Language (язык гипертекстовых приложений); по сути, это хорошо известные типы содержимого — XML и JSON — с дополнительными соглашениями по представлению гиперссылок. Больше узнать о стандарте HAL можно в статье Майка Келли (Mike Kelly) (http://stateless.co/hal_specification.html). Примеры в оставшейся части главы используют тип application/hal+json, поэтому далее в этой главе вы так или иначе познакомитесь с основами этого типа.

Другой тип содержимого, который можно заметить на диаграмме контейнеров, — это application/atom+xml, обозначающий формат Atom. Atom — это распространенный формат синдикации лент новостей и потому отлично подходит для представления списков событий. Он в точности соответствует потребностям ограниченного контекста управления учетными записями для представления списка событий «Последовать за единомышленником».

Использование формата Atom для представления ленты событий является основой распределенных систем REST, управляемых событиями, которые будут создаваться далее в этой главе. Конечно, необязательно было бы использовать именно Atom, но этот формат настолько популярен, что вам определенно стоит познакомиться с ним.

На рис. 13.12 изображен пример получения ленты Atom с предметными событиями. Эта диаграмма показывает последовательность сообщений HTTP, вовлеченных в получение ленты событий «Последовать за единомышленником», которая будет реализована далее в этой главе.

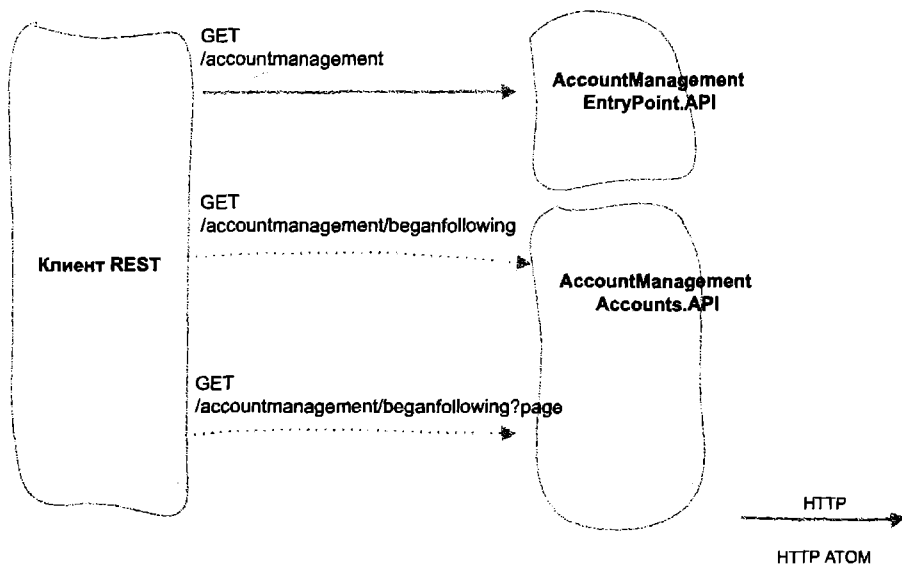


Рис. 13.12. Последовательность HTTP-запросов для получения ленты событий «Последовать за единомышленником»

Создание управляемых событиями систем REST с использованием ASP.NET Web API

Теперь, после знакомства с теорией, можно приступить к практическому воплощению RESTful-версии приложения социальной сети, управляемого событиями. Чтобы не усложнять примеры и помочь вам сосредоточиться на основных шаблонах, мы создадим лишь ограниченный контекст управления учетными записями и некоторый ограниченный контекст поиска. Из этих примеров вы узнаете достаточно, чтобы начать применять новые знания в своих проектах.

ПРИМЕЧАНИЕ

Мы реализуем не все ограниченные контексты, показанные на диаграммах в этом разделе, но наша главная цель не в этом, а в том, чтобы показать их архитектуру, чтобы вы понимали, как проектировать большие системы. Можете не сомневаться, имеющиеся примеры охватывают все важные понятия. А создание полной системы превратилось бы в ненужное повторение.

При создании системы RESTful социальной сети будет использовать прием разработки снаружи внутрь; сначала добавим точку входа в ограниченный контекст управления учетными записями. Затем создадим Accounts API. После этого реализуем ленту Atom для публикации событий «Последовать за единомышленником» (Began Following). И наконец, создадим потребителя ленты событий в ограниченном контексте поиска.

Весь программный код примеров, что следуют ниже, для удобства помещен в одно общее решение Visual Studio. Но вообще, при создании систем RESTful, которые интегрируются посредством HTTP, это необязательно. Фактически для реализации разных контекстов могут использоваться настолько разные технологии, что их нельзя объединить в одно решение. Вам выбирать, что лучше; использование разных репозиторий для отдельных частей системы способствует слабой связанности, тогда как хранение кода в одном месте может помочь составить общую картину.

Начать строительство новой RESTful-системы социальной сети можно с создания нового решения Visual Studio с именем PPPDDD.REST.SocialMedia.

Прикладные интерфейсы для гиперсреды

Гиперсреда — это основа REST. Поэтому далее будет показано, как сконструировать прикладной программный интерфейс для гиперсреды в .NET с помощью ASP.NET Web API. Детали реализации могут отличаться для разных фреймворков, но сама идея не зависит от выбора фреймворка и может применяться для реализации прикладных интерфейсов в гиперсреде с применением любых инструментов, какие вы выберете.

Как вы уже знаете, главным преимуществом гиперсреды является слабая связанность клиентов и серверов, что позволяет развивать их независимо. Но перед началом разработки клиентов необходимо иметь некоторую информацию о при-

кладном программном интерфейсе. Эту роль выполняет ресурс точки входа, к которому клиенты *должны* подключаться.

Ресурс точки входа

Когда возникает необходимость взаимодействий с REST API, клиент сначала запрашивает ресурс точки входа. Далее он просто следует по возвращаемым ему гиперссылкам. Выбирая местоположение точки входа, следует учитывать множество факторов. Например, в архитектуре, изображенной на рис. 13.10, было решено создать по одной точке входа для каждого ограниченного контекста. Но точно так же можно было бы создать одну точку входа для всей системы или, напротив, пойти по пути более мелкого дробления и создать точки входа для всех ресурсов верхнего уровня. В любом случае для каждого проекта необходимо принять решение, какие части системы экспортировать через ресурсы точек входа.

Проектируя точки входа, нужно определить, какие начальные и переходные ресурсы должны быть доступны потребителям прикладного интерфейса. Точка входа в ограниченный контекст управления учетными записями будет представлять собой список ресурсов верхнего уровня. В данном примере в этот список будет входить лишь один ресурс — Accounts. Из ресурса Accounts гиперссылки будут вести к отдельным учетным записям, а из отдельных учетных записей — к подробной информации о них, экспортируемой в виде дочерних ресурсов, таких как список последователей (followers). Это описание не содержит ничего нового — оно лишь повторяет то, что вы видели на диаграмме последовательности операций (см. рис. 13.11).

Как вы уже наверняка поняли, ссылки — это строительные блоки гиперсреды. Но чтобы клиенты могли следовать по ссылкам, они должны иметь возможность определять, какие ссылки представляют нужные им переходы. Эту роль играют ссылочные отношения, которые указывают, что именно представляют ссылки (или, выражаясь точнее, их отношение к текущему ресурсу). Например, чтобы ресурс с множеством страниц был совместим с гиперсредой, он должен иметь ссылочное отношение Next (Далее), которое клиенты могли бы использовать для перехода к следующей странице. В последующих примерах вы увидите множество ссылок и ссылочных отношений.

ПРИМЕЧАНИЕ

Больше узнать о типичных ссылочных отношениях можно в официальной документации администрации адресного пространства Интернета (Internet Assigned Numbers Authority, IANA), где содержится исчерпывающий список (<http://www.iana.org/assignments/link-relations/link-relations.xhtml>). Допускается также использовать собственные ссылочные отношения. В этом случае вы меняете выгоды от использования общеизвестных соглашений, понятных всем, на менее известные соглашения, зато более ясно отражающие предметную область.

Чтобы создать API, возвращающий точку входа в ограниченный контекст управления учетными записями, добавьте в решение новое приложение типа ASP.NET с именем AccountManagement.EntryPoint. Это имя соответствует соглашениям по

именованию проектов API, основанным на формате {ограниченный контекст}. {Ресурс}. Добавляя проект в решение, выберите пустой шаблон и установите флажок Web API.

ПРИМЕЧАНИЕ

На протяжении всего этого раздела при создании в Visual Studio всех проектов типа ASP.NET Web Application следует выбирать пустой шаблон и устанавливать флажок Web API.

Осталось принять еще одно решение, прежде чем можно будет добавить конечную точку, возвращающую ресурс точки входа. Вы должны выбрать тип MIME, поддерживаемый гиперсредой.

HAL

Традиционно форматом гиперсреды для REST API служит XHTML. К сожалению, использовать его нежелательно из-за избыточности (особенно если вы пытаетесь уйти от SOAP). К счастью, существует относительно новый и достаточно обкатанный стандарт. Этот новый стандарт — HAL, язык гипертекстовых приложений, упоминавшийся выше в этой главе и имеющий два основных диалекта: XML и JSON. По сути, HAL является расширением этих двух известных форматов, добавляющим определенные соглашения по представлению гиперссылок. Это позволяет пользоваться преимуществами гиперсреды языка XHTML без присущей ему избыточности, как демонстрирует следующий пример.

```
{
  "_links": {
    "self": {
      "href": "http://localhost:4100/accountmanagement"
    },
    "accounts": {
      "href": "http://localhost:4101/accounts"
    },
  }
}
```

Ресурс точки входа, показанный в предыдущем фрагменте, демонстрирует соглашения по представлению гиперссылок в языке HAL (JSON). Все ссылки должны определяться в корне ресурса, в элементе `_links`. Каждая ссылка начинается со своего ссылочного отношения (в данном примере `self` и `accounts`). Также каждая ссылка содержит атрибут `href` с идентификатором URI ресурса, на который она указывает. Ссылка `self`, указывающая на текущий ресурс, является единственной обязательной ссылкой; любые другие добавляются по мере необходимости.

Чтобы создать ресурс точки входа в API, нужно сначала настроить проект, чтобы после запуска он использовал порт 4100. (Установить этот параметр можно в свойствах проекта, во вкладке `Web`.) Затем следует настроить URI точки входа, добавив маршрут в файл `WebAPIConfig` проекта Web API, как показано в листинге 13.8. Этот файл находится в папке `App_Start`, в корне проекта.

Листинг 13.8. Добавление маршрута к точке входа в файле WebApiConfig

```
using System;
using System.Web.Http;

namespace AccountManagement.EntryPoint.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "Entry Point",
                routeTemplate: "accountmanagement",
                defaults: new { controller = "EntryPoint", action = "Get" }
            );
        }
    }
}
```

Для тех, кто не знаком с синтаксисом маршрутов Web API: в листинге 13.8 определяется маршрут Entry Point, гарантирующий, что любой запрос с адресом /accountmanagement будет обрабатываться методом Get() класса EntryPoint-Controller. Прежде чем приступить к реализации этого контроллера, нужно с помощью Nuget установить пакет, добавляющий поддержку HAL в Web API. Как вы увидите далее, это невероятно удобная библиотека, решающая все рутинные задачи, связанные с созданием HAL API. Чтобы установить webApi.hal в проект AccountManagement.EntryPoint.Api, выполните следующую команду в консоли диспетчера пакетов Nuget:

```
Install-Package WebApi.Hal -Project AccountManagement.EntryPoint.Api
```

ПРИМЕЧАНИЕ

Web API поддерживает соглашения, помогающие избежать необходимости явно определять каждый маршрут. В этом примере маршруты определены явно, чтобы сделать его более понятным. Но при желании вы можете выбрать другой подход.

После установки пакета WebApi.Hal нужно выбрать тип HAL (JSON) как тип MIME по умолчанию, указав его в файле Global.asax.cs, как показано в листинге 13.9. Также в этом листинге вторым, альтернативным типом MIME устанавливается тип HAL (XML). Два средства форматирования, JsonHalMediaTypeFormatter и XmlHalMediaTypeFormatter, находятся в только что установленном пакете WebAPI.Hal.

Листинг 13.9. Установка HAL в качестве типа содержимого по умолчанию

```
using System;
using System.Web.Http;
using WebApi.Hal;
```

```

namespace AccountManagement.EntryPoint.Api
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);

            // тип MIME по умолчанию (HAL JSON)
            GlobalConfiguration.Configuration.Formatters.Insert(
                0, new JsonHalMediaTypeFormatter()
            );

            // альтернативный тип MIME (HAL XML)
            // заголовок accept должен хранить строку "application/hal+xml"
            GlobalConfiguration.Configuration.Formatters.Insert(
                1, new XmlHalMediaTypeFormatter()
            );
        }
    }
}

```

Все зависимости установлены и настроены, теперь осталось проложить путь к прикладному интерфейсу точки входа, реализовав `EntryPointController`. Для этого добавьте класс с именем `EntryPointController` в папку `Controllers`, находящуюся в корне проекта. После этого замените содержимое файла программным кодом из листинга 13.10.

Листинг 13.10. Класс `EntryPointController` для проекта `AccountManagement.EntryPoint.Api`

```

using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.EntryPoint.Api.Controllers
{
    public class EntryPointController : ApiController
    {
        private const string EntryPointBaseUrl = "http://localhost:4100/";
        private const string AccountsBaseUrl = "http://localhost:4101/";

        [HttpGet]
        public EntryPointRepresentation Get()
        {
            return new EntryPointRepresentation
            {
                Href = EntryPointBaseUrl + "accountmanagement",
                Rel = "self",
                Links = new List<Link>
                {

```

```

        new Link
        {
            Href = AccountsBaseUrl + "accountmanagement/accounts",
            Rel = "accounts",
        },
    };
}

public class EntryPointRepresentation : Representation
{
    protected override void CreateHypermedia(){}
}
}

```

Код в листинге 13.10 возвращает ресурс точки входа в формате HAL-JSON (по умолчанию). Это объясняется тем, что `EntryPointRepresentation` наследует базовый класс `Representation` — класс, предоставляемый библиотекой `WebApi.Hal`. Когда метод контроллера возвращает класс, наследующий `Representation`, метод `JsonHalMediaTypeFormatter` преобразует его в формат HAL-JSON.

Код внутри `Get()` декларативно отображается в формат ответа. Здесь можно видеть, что код генерирует две ссылки: `Href` и `Links` (коллекция с единственной ссылкой). Эти две ссылки появятся в ответе. Увидеть, как все это действует, можно, протестировав то, что уже имеется.

Тестирование HAL API с помощью браузера HAL

Огромное преимущество строительства прикладных интерфейсов в гиперсреде на основе единых стандартов заключается в простоте создания типовых клиентов, с помощью которых можно исследовать такие прикладные интерфейсы. Одним из таких инструментов для HAL является браузер HAL, небольшое веб-приложение, позволяющее взаимодействовать с HAL API. Далее вы увидите, как с помощью браузера HAL протестировать API только что созданной точки входа.

Чтобы установить браузер HAL, загрузите архив (<https://github.com/mikekelly/halbrowser/archive/master.zip>) и распакуйте его. Скопируйте распакованное содержимое в корневую папку проекта `AccountManagement.EntryPoint.Api` с помощью Windows Explorer (Проводник Windows). В корневой папке проекта `AccountManagement.EntryPoint.Api` должен находиться файл с именем `browser.html`. Если вы его не нашли там, значит, файлы были скопированы вами не в то место.

Если вы не ошиблись с каталогом назначения при копировании файлов, нажмите клавишу F5, чтобы запустить проект. В веб-браузере, который Visual Studio запустит автоматически, откройте браузер HAL, указав адрес <http://localhost:4100/browser.html>. (Здесь предполагается, что проект был настроен на использование порта 4100, как описывалось выше.) После открытия страницы `browser.html`, если все работает правильно, должен появиться пользовательский интерфейс браузера HAL, как показано на рис. 13.13.

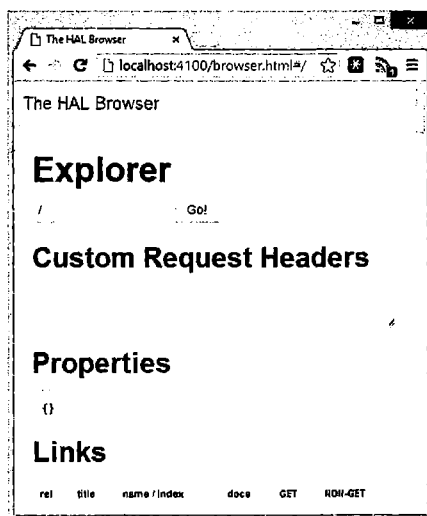


Рис. 13.13. Браузер HAL

ПРИМЕЧАНИЕ

Если возникнут какие-либо проблемы с браузером HAL или любыми другими задачами, описываемыми в этой главе, вы можете обратиться за помощью на дискуссионный форум Wrox по адресу <http://p2p.wrox.com/>. Также для справки можно обратиться к исходному коду в загружаемых примерах для этой главы.

Обратите внимание, что сразу после открытия браузера HAL он не отображает ресурс точки входа. Это объясняется тем, что браузер HAL ищет точки входа в API в пути по умолчанию (/). Чтобы исправить это, просто введите `/accountmanagement` в адресную строку браузера HAL (поле ввода ниже надписи Explorer (Обозреватель)) и щелкните на кнопке GO (Вперед). После этого на экране должна появиться информация о ресурсе точки входа (справа) и интерактивные инструменты (слева), как показано на рис. 13.14.

В настоящий момент браузер HAL не дает особых выгод, потому что ссылки в ресурсе точки входа указывают на несуществующие ресурсы. Поэтому перейдем к решению следующей задачи: реализуем прикладной интерфейс Accounts. Как указывает содержимое ресурса точки входа (рис. 13.14), ресурс Accounts должен быть доступен по адресу <http://localhost:4101/accountmanagement/accounts>.

Шаблоны URI

В процессе создания прикладного интерфейса Accounts вы узнаете, как бороться с такой типичной проблемой гиперсреды, как неэффективная навигация. Представьте API, экспортирующий массу данных, например тысячи и миллионы учетных записей. Клиентам может потребоваться просмотреть сотни ссылок, следуя по ссылкам Next (Далее), чтобы найти нужный ресурс. Очевидно, что это

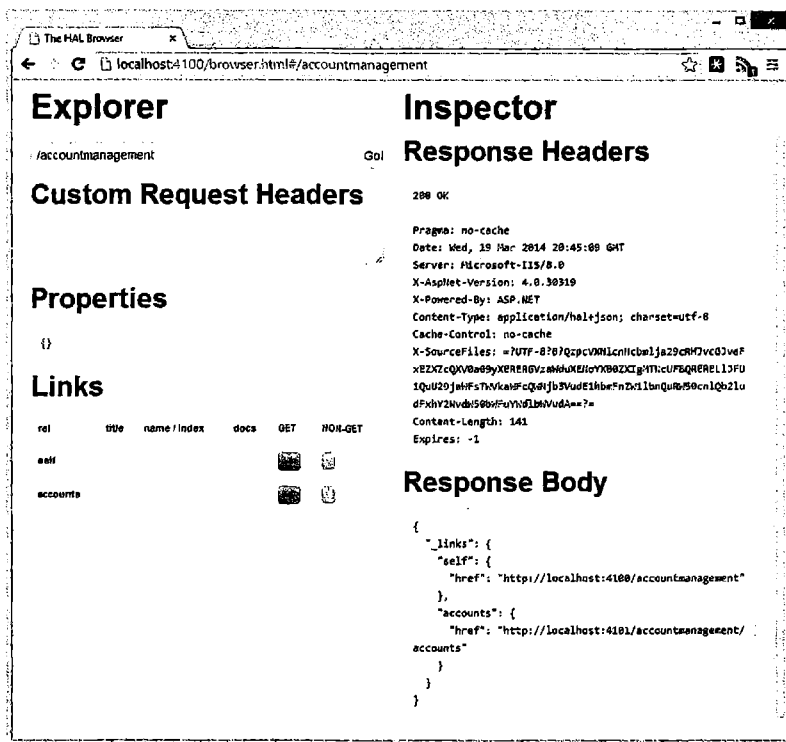


Рис. 13.14. Представление ресурса точки входа в браузере HAL

чудовищно неэффективно и для клиента, и для сервера, особенно в общедоступных API, обслуживающих одновременно большое число клиентов. Эта проблема решается с помощью шаблонов URI.

В следующем фрагменте демонстрируется ресурс Accounts в формате HAL (JSON), для которого вскоре мы создадим прикладной интерфейс. Взгляните на шаблон URI; это ссылка с атрибутом `templated`, установленным в значение `true`:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:4101/accountmanagement/accounts"
    },
    "alternative": {
      "href": "http://localhost:4101/accountmanagement/accounts?page=1"
    },
    "account": [
      {
        "href": "http://localhost:4101/accountmanagement/accounts/{accountId}",
        "templated": true
      },
      {

```

```

    "href": "http://localhost:4101/accountmanagement/accounts/123"
  },
  ...
}

```

Для определения шаблонов URI нужно не только присвоить атрибуту `templated` значение `true`, но также добавить в URI поле для подстановки. В определении ресурса `Accounts` таким полем для подстановки является `{accountId}`. Клиенты API могут заменять это поле идентификатором искомой учетной записи. Такой подход позволяет сократить сотни потенциальных запросов до одного. Создание шаблонов URI с применением `WebApi.Nal` требует приложить чуть больше усилий, в чем вы убедитесь в процессе создания прикладного интерфейса `Accounts`.

Чтобы создать прикладной интерфейс `Accounts`, добавьте новый проект с именем `AccountManagement.Accounts.Api` и настройте его, чтобы после запуска он использовал порт 4101. Затем добавьте в проект ссылку на `WebApi.Nal`, выполнив следующую команду в консоли диспетчера пакетов Nuget:

```
Install-Package WebApi.Nal -Project AccountManagement.Accounts.Api
```

Последний шаг в настройке проекта — выбор типа `NAL` в качестве типа содержимого по умолчанию, как показано выше в листинге 13.9.

Прикладной интерфейс `Accounts` будет экспортировать два URI. Прежде всего, клиенты будут обращаться к ресурсу `Accounts` по адресу `/accountmanagement/accounts`, представляющему список всех учетных записей. Отсюда клиенты будут переходить к отдельным учетным записям, используя шаблон URI в ресурсе `Accounts` — `accountmanagement/accounts/{accountId}`. Чтобы объявить эти маршруты в проекте, отредактируйте файл `WebApiConfig` в проекте `AccountManagement.Accounts.Api`, чтобы его содержимое выглядело, как показано в листинге 13.11.

Листинг 13.11. Объявление маршрутов для прикладного интерфейса `Accounts` в `WebApiConfig`

```

using System;
using System.Web.Http;

namespace AccountManagement.Accounts.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "Accounts Collection",
                routeTemplate: "accountmanagement/accounts",
                defaults: new { controller = "Accounts", action = "Index" }
            );

            config.Routes.MapHttpRoute(

```

```

        name: "Individual Account",
        routeTemplate: "accountmanagement/accounts/{accountId}",
        defaults: new { controller = "Accounts", action = "Account" }
    });
}
}
}
}
}

```

Как можно заметить в объявлениях маршрутов в листинге 13.11, в папку `Controllers` (внутри проекта `AccountManagement.Accounts.Api`) нужно добавить класс с именем `AccountsController`. Он должен иметь два метода — `Index()` и `Accounts()`, — как того требуют определения маршрутов. Начнем с метода `Index()`. Первоначально класс `AccountsController` должен содержать код, представленный в листинге 13.12.

Листинг 13.12. `AccountsController`

```

using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.Accounts.Api.Controllers
{
    public class AccountsController : ApiController
    {
        private const string EntryPointBaseUrl = "http://localhost:4100/";
        private const string AccountsBaseUrl =
            "http://localhost:4101/ accountmanagement/";

        [HttpGet]
        public AccountsRepresentation Index()
        {
            return new AccountsRepresentation
            {
                Href = AccountsBaseUrl + "accounts",
                Rel = "self",
                Links = new List<Link>
                {
                    new Link
                    {
                        Href = AccountsBaseUrl + "accounts?page=1",
                        Rel = "alternative",
                    },
                    new Link
                    {
                        // автоматически идентифицируется как шаблон
                        Href = AccountsBaseUrl + "accounts/{accountId}",
                        Rel = "account",
                    },
                    new Link
                    {
                        Href = AccountsBaseUrl + "accounts/123",

```

```

        Rel = "account",
    },
    new Link
    {
        Href = AccountsBaseUrl + "accounts/456",
        Rel = "account",
    },
    new Link
    {
        Href = AccountsBaseUrl + "accaccounts?page=2",
        Rel = "next"
    },
    new Link
    {
        Href = EntryPointBaseUrl + "accountmanagement",
        Rel = "parent"
    }
    },
};
}

public class AccountsRepresentation : Representation
{
    protected override void CreateHypermedia()
    {
    }
}
}

```

Большая часть кода в листинге 13.12 должна быть знакома по листингу 13.10, тем не менее есть смысл сделать несколько замечаний относительно дополнительных ссылок. Ссылочное отношение `alternative` представляет ссылки, имеющие отличающиеся URI, но указывающие на тот же ресурс (`self`). Благодаря таким ссылкам клиенты получают возможность организовать более эффективное кэширование, интерпретируя оба URI как один и тот же ресурс. Ниже ссылки `alternative` определяется шаблонная ссылка `account`, которую библиотека `WebApi.Hal` автоматически отметит как шаблонную, потому что ее атрибут `href` содержит поле для подстановки.

Прежде чем тестировать новый API в браузере HAL, нужно разрешить совместное использование ресурсов между разными источниками (Cross-Origin Resource Sharing, CORS), потому что ресурс `Accounts` находится на другом виртуальном хосте (порт 4101 вместо 4100). Подробное описание включения поддержки CORS можно найти на веб-сайте ASP.NET (www.asp.net/web-api/overview/security/enabling-cross-origin-requests-in-web-api)¹. В простейшем случае для этого нужно выполнить следующие шаги.

¹ Похожую статью на русском языке можно найти по адресу <https://msdn.microsoft.com/ru-ru/magazine/dn532203.aspx>. — Примеч. пер.

1. Добавить в проект пакет CORS, выполнив следующую команду в консоли диспетчера пакетов Nuget:

```
Install-Package Microsoft.AspNet.WebApi.Cors -Project AccountManagement.  
Accounts.Api
```

2. Настроить поддержку CORS в WebApiConfig, как показано в листинге 13.13.

Листинг 13.13. Включение поддержки CORS в ASP.NET WEB API

```
using System;  
using System.Web.Http;  
using System.Web.Http.Cors;  
  
namespace AccountManagement.Accounts.Api  
{  
    public static class WebApiConfig  
    {  
        public static void Register(HttpConfiguration config)  
        {  
            var cors = new EnableCorsAttribute("http://localhost:4100", "*", "*");  
            config.EnableCors(cors);  
  
            // остальной код
```

Если теперь запустить браузер HAL, как описывалось выше, перейти по адресу ресурса точки входа и проследовать по ссылке до ресурса Accounts, вы увидите URI шаблонной ссылки, как показано на рис. 13.15.

ВНИМАНИЕ

В момент запуска приложения после включения поддержки CORS вы можете увидеть сообщение об ошибке, извещающее, что версия `System.Web.Http` не может быть загружена. Чтобы исправить эту проблему, нужно настроить привязку сборок. Когда вы будете компилировать решение, после появления предупреждения, начинающегося со слов `Found conflicts between...` (в панели `Error List` (Список ошибок)), щелкните на нем дважды, и Visual Studio автоматически добавит необходимые привязки.

Шаблоны URI допускают возможность объединения их с обычными ссылками, даже указывающими на тот же ресурс. На рис. 13.15 можно видеть две нешаблонные ссылки «account», иллюстрирующие такую возможность. Они указывают непосредственно на ресурсы Account. Однако чтобы эти ссылки могли действовать, нужно добавить метод `Account()` в класс `AccountsController`. (Это определяется объявлением маршрута в листинге 13.11.)

В листинге 13.14 представлена начальная реализация `Account()`, возвращающая фиксированные данные. Вы можете добавить ее в класс `AccountsController`, ниже определения метода `Index()`. Вам также нужно добавить классы `Account-Representation` и `Account` из листинга 13.15. (Оба можно вставить в файл `AccountsController.cs`.)

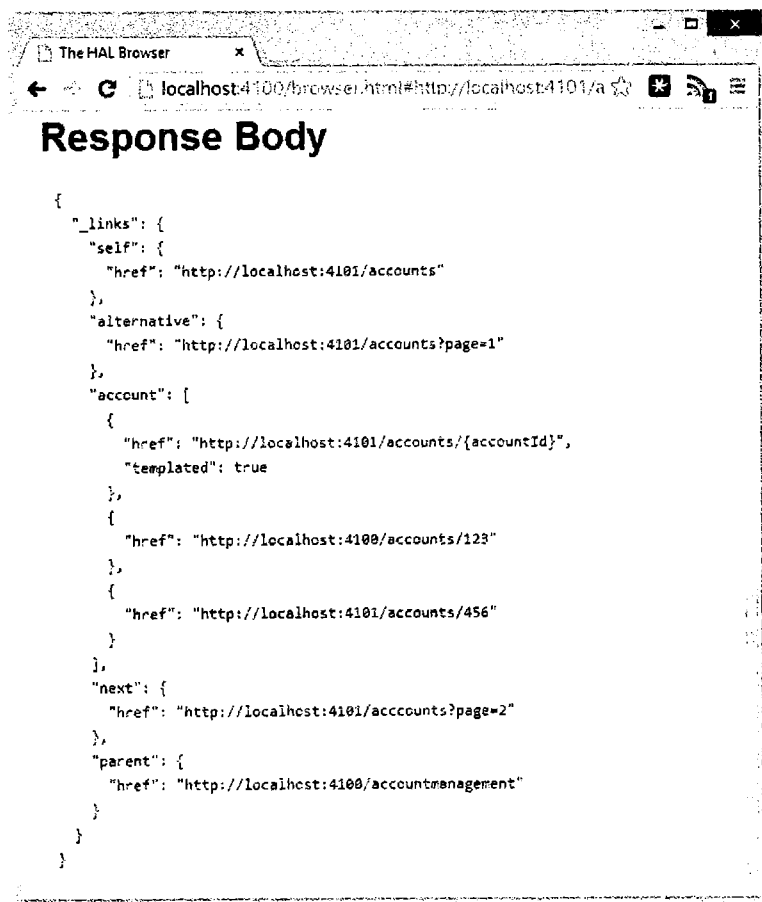


Рис. 13.15. Результат перехода по ссылке к ресурсу Accounts в браузере HAL

Листинг 13.14. Метод Account, генерирующий ресурсы типа Account

```

[HttpGet]
public AccountRepresentation Account(string accountId)
{
    // фиксированные данные.. пока
    return new AccountRepresentation
    {
        Href = AccountsBaseUrl + "accounts/" + accountId,
        Rel = "self",
        AccountId = accountId,
        Name = "Account_" + accountId,
        Links = new List<Link>
        {
            new Link
            {
                Href = AccountsBaseUrl + "accounts",

```

```

        Rel = "collection",
    },
    new Link
    {
        Href = AccountsBaseUrl + "accounts/" + accountId + "/followers",
        Rel = "followers",
    },
    new Link
    {
        Href = AccountsBaseUrl + "accounts/" + accountId + "/following",
        Rel = "following",
    },
    new Link
    {
        Href = AccountsBaseUrl + "accounts/" + accountId + "/blurbs",
        Rel = "blurbs",
    }
}
};
}

```

Листинг 13.15. Класс Account Representation, используемый для создания ресурса Account

```

public class AccountRepresentation : Representation
{
    public string AccountId { get; set; }
    public string Name { get; set; }

    protected override void CreateHypermedia()
    {
    }
}

```

Поля ресурса (в противоположность ссылкам) представляются в HTTP-ответе в виде текста в стандартном формате JSON. В листинге 13.15 можно видеть, что AccountRepresentation имеет свойства AccountId и Name. Соответственно после перехода к ресурсу Account в браузере HAL вы увидите эти свойства, представленные в виде текста JSON, как показано на рис. 13.16.

На рис. 13.16 также можно видеть некоторые особенно примечательные ссылки. В частности, три ссылки, указывающие на дочерние ресурсы данной учетной записи: followers, following и blurbs. Конечная точка followers будет реализована в следующем разделе, но другие две ссылки приведены здесь, только чтобы показать, как может выглядеть ресурс с множеством дочерних ресурсов. С созданием конечной точки followers мы вступаем в управляемые событиями части системы, которые будут реализованы с применением Event Store¹.

¹ Как указано на сайте проекта <https://geteventstore.com/>, Event Store — это «открытая, эффективная база данных, поддерживающая сложную обработку событий в JavaScript». — *Примеч. пер.*

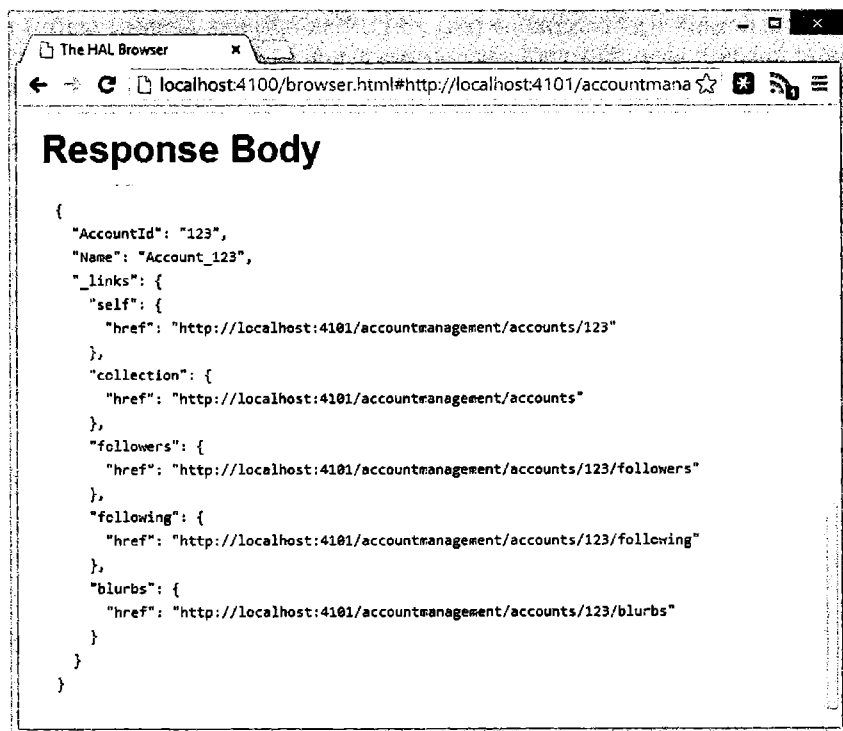


Рис. 13.16. Поля данных ресурса, представленные в виде текста в формате JSON

Сохранение событий с помощью Event Store

Мы создали большую часть инфраструктуры и теперь готовы перейти к изучению приемов создания REST-систем, управляемых событиями. В первой части последующего описания рассказывается о сохранении событий. Существует много способов, позволяющих сделать это, например записывать события в текстовый файл или использовать таблицу в базе данных SQL. Но в данном примере вы познакомитесь со специализированным инструментом — Event Store, — созданным известным практиком DDD Греггом Янгом и его соратниками (www.geteventstore.com).

ПРИМЕЧАНИЕ

Вам может быть полезно еще раз взглянуть на диаграммы, изображенные на рис. 13.9 — 13.11, чтобы вспомнить, как объединяются все части, созданные к настоящему моменту, и каковы их цели. Так просто потерять общее представление, занимаясь низкоуровневыми деталями.

Чтобы познакомиться с особенностями сохранения событий, нужно создать в прикладном интерфейсе Accounts конечную точку, возвращающую последователей (followers) данной учетной записи. Эта конечная точка поддерживает возможность добавления новых последователей в коллекцию. Последовательность необходимых

для этого действий изображена на рис. 13.11. Как обычно, первым шагом на пути создания новой конечной точки, экспортирующей ресурсы, является определение маршрута. В листинге 13.16 показано, что нужно включить в файл `WebApiConfig`, чтобы добавить определение маршрута для конечной точки `Followers`.

Листинг 13.16. Измененный файл `WebApiConfig` с определением маршрута для `Followers`

```
using System;
using System.Web.Http;

namespace AccountManagement.Accounts.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            // ... прежние определения маршрутов остаются без изменений

            // определение нового маршрута, которое требуется добавить
            config.Routes.MapHttpRoute(
                name: "Account Followers",
                routeTemplate: "accountmanagement/accounts/{accountId}/followers",
                defaults: new { controller = "Followers", action = "Index" }
            );
        }
    }
}
```

В листинге 13.16 показано, что ресурс `Followers` будет обслуживаться методом `Index()` в контроллере `FollowersController`. Чтобы создать этот контроллер, добавьте в папку `Controllers` проекта класс с именем `FollowersController` и с реализацией в листинге 13.17.

Листинг 13.17. Начальная реализация `FollowersController`

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.Accounts.Controllers
{
    public class FollowersController : ApiController
    {
        private const string AccountsBaseUrl =
            "http://localhost:4101/accountmanagement/";

        [HttpGet]
        public FollowersRepresentation Index(string accountId)
        {
            return new FollowersRepresentation
            {

```

```

        Href = AccountsBaseUrl + "accounts/" + accountId + "/followers",
        Rel = "self",
        Links = new List<Link>
        {
            new Link
            {
                Href = AccountsBaseUrl +
                    "accounts/" + accountId + "/followers?pages=2",
                Rel = "next",
            },
        },
        followers = GetFollowers(accountId)
    };
}

private List<Follower> GetFollowers(string accountId)
{
    // заменить поиском в базе данных, например (в действующем приложении)
    return new List<Follower>
    {
        new Follower
        {
            AccountId = "f11",
        },
        new Follower
        {
            AccountId = "f12",
        },
        new Follower
        {
            AccountId = "f13",
        }
    };
}

public class FollowersRepresentation : Representation
{
    public List<Follower> followers { get; set; }

    protected override void CreateHypermedia()
    {
    }
}

public class Follower
{
    public string AccountId { get; set; }
}
}

```

Реализация метода `Index()` в листинге 13.17 просто возвращает фиксированный ответ, параметризованный указанным идентификатором учетной записи. Это не самая важная часть в данном примере — гораздо более важной частью является

сохранение события с данными, которые были переданы в конечную точку. Ее реализация приводится в листинге 13.18, содержимое которого следует добавить в класс `FollowersController`, сразу вслед за определением метода `GetFollowers()`. Также нужно добавить класс `BeganFollowing`, который показан в листинге 13.19.

Листинг 13.18. Сохранение события в `FollowersController` по запросу POST

```
[HttpPost] // вызывается только в ответ на запросы POST
[ActionName("index")] // Web API не позволяет использовать повторяющиеся имена
public IHttpActionResult IndexPOST(string accountId, Follower follower)
{
    // accountId извлекается из строки запроса – это простой тип
    // follower извлекается из тела запроса – это составной тип

    var evnt = new BeganFollowing
    {
        AccountId = accountId,
        FollowerId = follower.AccountId
    };
    EventPersister.PersistEvent(evnt);
    return RedirectToRoute("Account Followers", new { accountId = accountId });
}
```

Листинг 13.19. Класс для представления предметного события «Последовать за единомышленником»

```
// представляет предметное событие
public class BeganFollowing
{
    public string AccountId { get; set; }

    public string FollowerId { get; set; }
}
```

ВНИМАНИЕ

Привязка модели к предметным классам или структурам данных, используемым для сохранения, может привести к образованию тесной связи. Представленное здесь решение выбрано для простоты и не рекомендуется для использования в действующих приложениях.

ПРИМЕЧАНИЕ

Конечная точка `Followers` была добавлена в проект `AccountManagement.Accounts.Api` для удобства. В зависимости от требований к масштабируемости вам может понадобиться выделить дочерние ресурсы в отдельные проекты, чтобы их можно было масштабировать независимо.

Метод `IndexPOST()` из листинга 13.18 вызывается в ответ на запросы POST по адресу `/accountmanagement/accounts/{accountId}/followers`. Это можно определить по атрибуту `HttpPost`. Поскольку в том же файле уже существует метод с именем `Index()`, атрибут `ActionName` указывает, что для маршрута "Account Followers" должен вызываться данный метод, даже при том что его имя `IndexPOST()`.

Механика сохранения событий не показана в листинге 13.18. Однако здесь можно видеть вызов `EventPersister.PersistEvent()`. Этот класс осуществляет сохранение события. Его определение приводится в листинге 13.20, где демонстрируется минимально необходимая функциональность, необходимая для сохранения событий в Event Store. Добавьте этот класс в свой проект. Для удобства его можно поместить в конец файла `AccountsController.cs` (но за пределами определения класса `AccountsController`). Однако прежде чем добавить `EventPersister`, нужно установить клиент Event Store для C#, выполнив следующую команду:

```
Install-Package EventStore.Client -Project AccountManagement.Accounts.Api
```

Листинг 13.20. `EventPersister` — вспомогательный класс для сохранения событий в Event Store

```
public static class EventPersister
{
    private static IPEndPoint defaultEsEndpoint =
        new IPEndPoint(IPAddress.Loopback, 1113);

    private static IEventStoreConnection esConn =
        EventStoreConnection.Create(defaultEsEndpoint);

    static EventPersister()
    {
        esConn.Connect();
    }

    public static void PersistEvent(object ev)
    {
        var commitHeaders = new Dictionary<string, object>
        {
            {"CommitId", Guid.NewGuid()},
        };

        esConn.AppendToStream(
            "BeganFollowing", ExpectedVersion.Any, ToEventData(Guid.NewGuid(),
            ev, commitHeaders
        ));
    }

    private static EventData ToEventData(Guid eventId, object evnt,
        IDictionary<string, object> headers)
    {
        var data = Encoding.UTF8.GetBytes(
            JsonSerializer.SerializeToString(evnt)
        );

        var metadata = Encoding.UTF8.GetBytes(
            JsonSerializer.SerializeToString(headers)
        );

        var typeName = evnt.GetType().Name;

        return new EventData(eventId, typeName, true, data, metadata);
    }
}
```

Чтобы класс `EventPersister` скомпилировался без ошибок, нужно добавить в файл следующие инструкции `using`:

```
using EventStore.ClientAPI;  
using Newtonsoft.Json;  
using System.Net;  
using System.Text;
```

В листинге 13.20 есть две важные детали, на которые следует обратить внимание: событие преобразуется в формат JSON (и затем в двоичный формат) и передается в поток, в данном случае в поток `BeganFollowing`. Вам проще будет понять, как все это работает, когда вы установите и запустите Event Store.

ПРИМЕЧАНИЕ

Регистрация событий — обширная и очень интересная тема. Она подробно рассматривается в главе 22 «Регистрация событий».

Установка и запуск Event Store

В этом примере используется версия Event Store 2.0.1 (<http://download.geteventstore.com/binaries/EventStore-OSS-Win-v2.0.1.zip>). После загрузки файла архива распакуйте его и выполните следующую команду в консоли PowerShell (с правами администратора):

```
./EventStore.SingleNode.exe --db .\ESData
```

Чтобы убедиться, что база данных Event Store успешно запустилась, попробуйте открыть веб-приложение администрирования, которое должно быть доступно по адресу <http://localhost:2113>. В случае успеха откроется страница с приветствием, изображенная на рис. 13.17. Если вы не увидели эту страницу, проверьте еще раз, запускали ли вы консоль PowerShell с правами администратора. Также посмотрите,



Рис. 13.17. Пользовательский интерфейс администрирования Event Store

не выводилось ли сообщений об ошибках в консоли PowerShell. Если страница с приветствием появилась, значит, база данных Event Store запущена и терпеливо ждет, когда ей будет предложено сохранить события.

ПРИМЕЧАНИЕ

За дополнительной информацией об Event Store, включая описание установки в разных окружениях, таких как Mono и EC2, лучше всего обратиться к официальной документации на сайте GitHub (<https://github.com/eventstore/eventstore/wiki/Running-the-Event-Store#on-windows-and-net>).

ПРИМЕЧАНИЕ

База данных Event Store автоматически завершает работу при выходе из Windows и не перезапускается автоматически. Если вы будете опробовать описываемые здесь примеры в разных сеансах, вам придется вызывать команду запуска Event Store, что приводилась выше, при каждом входе в Windows.

Просмотр хранимых событий с помощью пользовательского интерфейса администратора Event Store

Ранее вы бегло ознакомились с пользовательским интерфейсом администратора Event Store, а теперь мы исследуем некоторые наиболее важные его возможности, которые помогут увидеть события, созданные прикладным интерфейсом Accounts. Чтобы добавить события в хранилище, нужно послать информацию о новых последователях. Для демонстрации все это можно сделать с помощью браузера HAL, открыв ресурс точки входа (/accountmanagement в браузере HAL) после запуска системы. Далее нужно проследовать по ссылке до ресурса Accounts. Из ресурса Accounts проследовать к одной из фиктивных учетных записей и оттуда проследовать по ссылке к ее ресурсу followers.

Чтобы послать запрос POST ресурсу followers, нужно щелкнуть на оранжевой кнопке в столбце NON-GET, в строке, представляющей ссылку self. Эта кнопка показана на рис. 13.18. Щелчок на этой кнопке откроет диалог, где можно ввести текст в формате JSON для отправки в конечную точку. На рис. 13.19 показан правильно отформатированный текст JSON с информацией о новом последователе, введенный в диалог. После ввода текста нужно щелкнуть на кнопке Make Request (Выполнить запрос), и он будет отправлен ресурсу.

Если в ответ на запрос был получен код состояния 200, сохраненное событие можно будет просмотреть с помощью пользовательского интерфейса администратора Event Store. Откройте страницу <http://localhost:2113/> и выберите в меню пункт

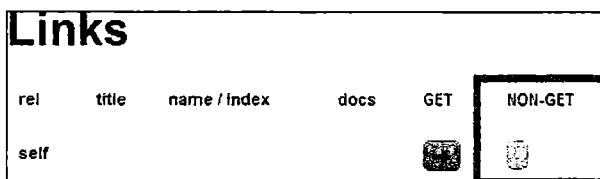


Рис. 13.18. Кнопка NON-GET в браузере HAL

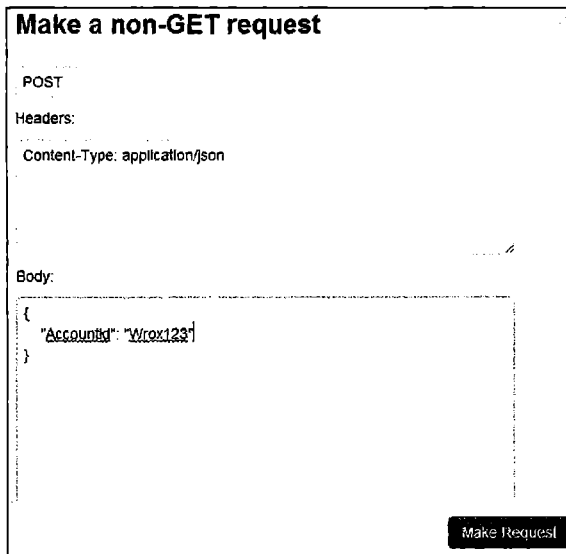


Рис. 13.19. Текст в формате JSON в диалоге NON-GET в браузере HAL

Streams (Потоки) — вы должны увидеть поток с названием **BeganFollowing**, созданный в результате передачи запроса POST из браузера HAL. Вы можете щелкнуть на названии потока, чтобы увидеть события, хранящиеся в нем. Во вновь открывшейся странице вы сможете исследовать отдельные события, как показано на рис. 13.20.

ПРИМЕЧАНИЕ

При попытке воспользоваться некоторыми функциями Event Store в веб-интерфейсе вам будет предложено ввести пароль. По умолчанию в Event Store используется имя пользователя **admin** и пароль **changeit**, как описывается в Вики Event Store (<https://github.com/EventStore/EventStore/wiki/Default-Credentials>).

Публикация событий в ленте Atom

Atom широко используется в системах RESTful для экспортирования событий, потому что это весьма распространенный формат, как уже говорилось ранее в этой главе. В данном примере вы узнаете, как использовать инструмент, встроенный в .NET и предназначенный для создания и публикации лент Atom.

Приложения, публикующие события в виде лент Atom, сродни компонентам, осуществляющим публикацию сообщений в системах обмена сообщениями. Соответственно для систем REST, управляемых событиями, можно использовать те же соглашения об именовании, передающие предметные понятия, например: {ОграниченныйКонтекст}. {БизнесКомпонент}. {Компонент}.

Чтобы создать компонент, публикующий событие «Последовать за единомышленником», можно добавить новое приложение типа ASP.NET Web Application в проект с именем **AccountManagement.RegularAccounts.BeganFollowing**. Настройте это приложение так, чтобы оно использовало порт 4102, и сделайте его запускаемым проектом.

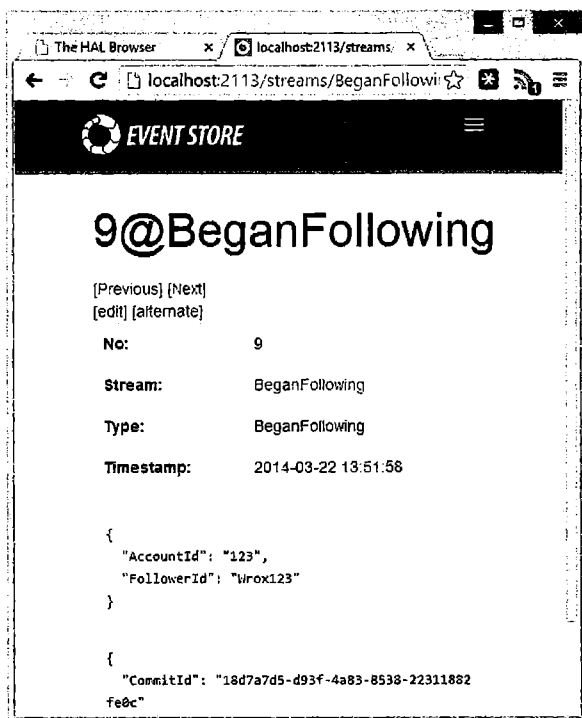


Рис. 13.20. Обзор событий, хранящихся в Event Store

Создание простой ленты Atom в .NET

Чтобы создать ленту Atom с использованием стандартных библиотек, входящих в состав фреймворка .NET, сначала нужно добавить ссылку на `System.ServiceModel` в новый проект `AccountManagement.RegularAccounts.BeganFollowing`. Затем следует определить маршрут, как показано в листинге 13.21, и после этого можно будет использовать классы из `System.ServiceModel` для создания ленты Atom, в которой будут публиковаться события из Event Store, как показано в листинге 13.22. Программный код из листинга 13.22 нужно добавить как новый контроллер в папку `Controllers`.

Листинг 13.21. Определение маршрута Began Following

```
config.Routes.MapHttpRoute(
    name: "Began Following",
    routeTemplate: "accountmanagement/beganfollowing",
    defaults: new { controller = "BeganFollowing", action = "Feed" }
);
```

Листинг 13.22. Контроллер BeganFollowing, генерирующий ленту RSS

```
using System;
using System.Net;
using System.Net.Http;
using System.Web.Http;
```

```
using System.ServiceModel.Syndication;
using System.Xml;
using System.Text;
using System.IO;
using EventStore.ClientAPI;
using Newtonsoft.Json;
using System.Collections.Generic;
using System.Linq;

namespace AccountManageent.RegularAccounts.BeganFollowing
{
    public class BeganFollowingController : ApiController
    {
        private const string BeganFollowingBaseUrl = "http://localhost:4102/";

        [HttpGet]
        public HttpResponseMessage Feed()
        {
            // создать ленту
            var feedUri = new Uri(BeganFollowingBaseUrl + "beganfollowing");
            var feed = new SyndicationFeed(
                "BeganFollowing", "Began following domain events", feedUri
            );
            feed.Authors.Add(
                new SyndicationPerson("accountManagementBC@wroxPPPPDD.com")
            );
            feed.Items = EventRetriever.RecentEvents("BeganFollowing")
                .Select(MapToFeedItem);

            // установить ленту как ответ - всегда atom+xml - не HAL
            var response = new HttpResponseMessage(HttpStatusCode.OK);
            response.Content = new StringContent(
                GetFeedContent(feed), Encoding.UTF8, "application/atom+xml"
            );
            return response;
        }

        private string GetFeedContent(SyndicationFeed feed)
        {
            using(var sw = new StringWriter())
            using(var xw = XmlWriter.Create(sw))
            {
                feed.GetAtom10Formatter().WriteTo(xw);
                xw.Flush();

                return sw.ToString();
            }
        }

        private SyndicationItem MapToFeedItem(ResolvedEvent ev)
        {
            return new SyndicationItem(
                "BeganFollowingEvent",
                Encoding.UTF8.GetString(ev.Event.Data),
                new Uri(
                    RequestContext.Url.Content("/beganfollowing/" +

```

```

        ev.Event.EventId
    )),
    ev.Event.EventId.ToString(),
    DateTime.Now // Клиент Event Store пока не возвращает отметку
                  времени
    );
}
}
}
}
}

```

Как можно заметить в листинге 13.22, лента Atom создается с помощью класса `SyndicationFeed`. Затем созданная лента устанавливается как ответ на запрос HTTP с помощью `XmlWriter`. Тип содержимого в объекте ответа указывается как `application/atom+xml`. Это значение будет записано непосредственно в HTTP-заголовок `Content-Type` ответа. В листинге можно также видеть, что отдельные события, извлекаемые из Event Store (`EventRetriever.RecentEvents()`), преобразуются в объекты `FeedItems`. Но в листинге 13.22 не показано, как извлекать события из Event Store. Об этом рассказывается далее.

Извлечение событий из Event Store

В листинге 13.22 отдельные элементы ленты генерируются на основе событий, извлекаемых из Event Store с помощью нестандартного класса `EventRetriever`. Определение этого класса приводится в листинге 13.23, и его следует добавить в проект. Чтобы избежать лишних сложностей с созданием нового файла, его можно просто поместить в конец файла с контроллером `BeganFollowingController`.

Листинг 13.23. `EventRetriever` — небольшой вспомогательный класс для извлечения событий из Event Store

```

public static class EventRetriever
{
    private static IPEndPoint defaultEsEndpoint =
        new IPEndPoint(IPAddress.Loopback, 1113);

    private static IEventStoreConnection esConn =
        EventStoreConnection.Create(defaultEsEndpoint);

    static EventRetriever()
    {
        esConn.Connect();
    }

    public static IEnumerable<ResolvedEvent> RecentEvents(string stream)
    {
        var results = esConn.ReadStreamEventsForward(stream, 0, 20, false);
        return results.Events;
    }
}

```

`EventRetriever` — это вспомогательный класс, служащий оберткой вокруг клиента Event Store для C#. В коде жестко определено, что извлекаться должно не более 20 последних событий, начиная с самого последнего. Извлечение выпол-

няется с помощью метода `ReadStreamEventsForward`, который начинает с самого последнего события и движется в обратном направлении. Клиент Event Store для C# обладает массой возможностей, которые не демонстрируются в этой книге, поэтому если вы решите использовать Event Store и клиента для C#, обращайтесь на веб-сайт Event Store, где можно найти много полезной информации.

Чтобы избежать ошибок во время компиляции `EventRetriever`, нужно добавить в проект ссылку на клиент Event Store для C#. Установка клиента выполняется следующей командой:

```
Install-Package EventStore.Client -Project AccountManagement.RegularAccounts.  
BeganFollowing
```

Чтобы убедиться, что лента Atom создается в соответствии с ожиданиями, сначала нужно дополнить реализацию ресурса точки входа (в проекте `AccountManagement.EntryPoint.Api`), добавив ссылку на ленту Atom (не забывайте, что клиенты должны обращаться к конкретным ресурсам только через точку входа, чтобы избежать образования тесных связей). В листинге 13.24 показано обновленное определение ресурса точки входа с требуемой ссылкой. Чтобы проверить ленту после добавления ресурса в проект, можно обратиться к ней непосредственно из браузера (прямое обращение к ресурсу во время тестирования — обычная практика): <http://localhost:4102/accountmanagement/beganfollowing>.

Листинг 13.24. Ресурс точки входа со ссылкой на ленту `BeganFollowing`

```
return new EntryPointRepresentation  
{  
    Href = EntryPointBaseUrl + "accountmanagement",  
    Rel = "self",  
    Links = new List<Link>  
    {  
        new Link  
        {  
            Href = AccountsBaseUrl + "accountmanagement/accounts",  
            Rel = "accounts",  
        },  
        new Link  
        {  
            Href = "http://localhost:4102/accountmanagement/beganfollowing",  
            Rel = "beganfollowing"  
        }  
    }  
};
```

Архивирование лент

В высокомасштабируемой системе, потенциально способной обслуживать миллионы пользователей, каждую секунду могут возникать сотни и тысячи событий. Использование единственной полосы Atom для всех этих событий может быстро сделать такую систему неработоспособной или привести к напрасному расходованию значительной доли ширины интернет-канала и другим проблемам, связанным с низкой эффективностью. Типичное решение заключается в том, чтобы ограничить число событий в каждой полосе и по достижении граничного значения

выполнять архивирование. Важно, чтобы каждая полоса содержала гиперссылки на предыдущий и следующий архивы (если имеются). Дополнительную информацию можно найти в документе RFC с названием «Feed Paging and Archiving» (<https://tools.ietf.org/html/rfc5005>), опубликованном рабочей группой по стандартам для сети Интернет Internet Engineering Task Force (IETF).

Создание подписчика на события/потребителя ленты Atom

Потребление ленты Atom с предметными событиями подобно подписке на сообщения в системах обмена сообщениями. Однако в случае с лентами Atom подписчик сам должен опрашивать ленту и извлекать из нее события. Конечно, такое решение требует от разработчиков приложить чуть больше усилий, но имеет и свои преимущества.

Создавая потребителя полосы Atom, можно опять воспользоваться преимуществом широкой популярности Atom в виде стандартных библиотек, входящих в состав фреймворка .NET. Чуть ниже вы увидите, как создать первую часть ограниченного контекста поиска, опрашивающую полосу с событиями «Последовать за единомышленником». Проект, решающий эту задачу, не обязательно должен быть веб-проектом. Вместо этого можно создать библиотеку классов C# с именем `Discovery.Recommendations.Followers` (в соответствии с диаграммой контейнеров на рис. 13.10). Как и прежде, чтобы воспользоваться библиотеками .NET, по-настоящему упрощающими синдицирование (RSS), нужно добавить ссылку в `System.ServiceModel`. Кроме того, этот проект необходимо сделать запускаемым.

ПРИМЕЧАНИЕ

Если вы не пользуетесь .NET, для вашей платформы также наверняка найдутся библиотеки или открытые проекты, упрощающие создание полос Atom. Для Java и, соответственно, Scala, например, имеется библиотека Rome Tools library (<http://rometools.github.io/rome/RssAndAtOMUtilitiesROMEVO.5AndAboveTutorialsAndArticles/RssAndAtOMUtilitiesROMEVO.5TutorialUsingROMEToCreateAndWriteASyndicationFeed.html>).

Подписка на события путем опроса

Логика опроса внутри нового компонента состоит из нескольких типичных шагов. Практически любое приложение, опрашивающее полосы Atom, будет выполнять похожие шаги.

1. Извлечь пакет событий, начиная с события, следующего за последним обработанным (или с первого, если до сих пор ни одно событие еще не обрабатывалось).
2. Обработать каждое событие в пакете в соответствии с правилами предметной области.
3. Сохранить идентификатор последнего обработанного события.

Первая часть реализации потребителя для ограниченного контекста поиска показана в листинге 13.25. Она содержит только логику верхнего уровня. Вы можете поместить весь этот код в единственный класс с именем `BeganFollowingPolling-FeedConsumer` в корень проекта.

Листинг 13.25. Логика опроса для BeganFollowingFeedConsumer

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Discovery.Followers.BeganFollowingConsumer
{
    public class BeganFollowingPollingFeedConsumer
    {
        const string EntryPointUrl = "http://localhost:4100/accountmanagement";

        private static string LastEventIdProcessed;

        public static void Main(String[] args)
        {
            // не лучшая реализация для использования в промышленных системах
            while(true)
            {
                FetchAndProcessNextBatchOfEvents();
                Thread.Sleep(1000);
            }
        }

        private static void FetchAndProcessNextBatchOfEvents()
        {
            var atomFeed = FetchFeed();
            var ups = GetUnprocessedEvents(atomFeed.Items.ToList());

            if (ups.Any())
                ProcessEvents(ups);
            else
                Console.WriteLine("No new events to process");
        }
    }
}

```

Листинг демонстрирует первую часть реализации потребителя полосы. Здесь показано, как можно извлечь и обработать пакет событий из полосы. Как видите, опрос выполняется не чаще одного раза в секунду, о чем свидетельствует вызов `Thread.Sleep()`.

Теперь уделим внимание низкоуровневым деталям — фактическому извлечению событий из полосы. Взгляните на листинг 13.26 — это пример клиента REST API, следующего по гиперссылкам от точки входа к целевому ресурсу. Этот код нужно добавить сразу вслед за кодом из листинга 13.25. Также добавьте в проект пакет `ServiceStack.Text`, выполнив следующую команду:

```
Install-Package ServiceStack.Text -Project Discovery.Recommendations.Followers
```

Листинг 13.26. Клиент REST API, следующий к полосе Atom по гиперссылкам

```

private static SyndicationFeed FetchFeed()
{
    using(var wc = new WebClient())
    {

```

```

// получить URI полосы из ресурса точки входа
var rawEp = wc.DownloadString(EntryPointUrl);
var hal = JsonSerializer.DeserializeFromString<HALResource>(rawEp);
var feedUrl = hal._links["beganfollowing"].href;

// выполнить парсинг строго типизированной полосы с событиями
var rawFeed = wc.DownloadString(feedUrl);
var feedXmlReader = XDocument.Parse(rawFeed).CreateReader();
return SyndicationFeed.Load(feedXmlReader);
};
}

```

Код в листинге 13.26 также зависит от классов в листинге 13.27 и следующих ниже инструкций `using`, которые нужно добавить в тот же файл:

Листинг 13.27. Модели клиента HAL

```

public class HALResource
{
    public Dictionary<string, Link> _links { get; set; }
}

public class Link
{
    public string href { get; set; }
}

```

После извлечения полосы можно приступить к обработке отдельных событий. Демонстрационная реализация такой обработки для класса `BeganFollowingPollingFeedConsumer` показана в листинге 13.28.

Листинг 13.28. Обработка событий и их маркировка как обработанных

```

private static List<SyndicationItem> GetUnprocessedEvents(
    List<SyndicationItem> events)
{
    var lastProcessed = events.SingleOrDefault(s => s.Id == LastEventIdProcessed);
    var indexOfLastProcessedEvent = events.IndexOf(lastProcessed);

    return events.Skip(indexOfLastProcessedEvent + 1).ToList();
}

private static void ProcessEvents(List<SyndicationItem> events)
{
    foreach (var ev in events)
    {
        var evnt = ParseEvent(ev.Content);
        Console.WriteLine(
            "Processing event: " + evnt.AccountId + " - " + evnt.FollowerId
        );

        // Здесь должна находиться реализация предметных правил
        LastEventIdProcessed = ev.Id;
    }
}

```

Листинг 13.28 демонстрирует типичную логику верхнего уровня, которую часто можно встретить в программах, потребляющих ленты. Сначала из ленты извлекается пакет событий, затем выбираются те, что еще не были обработаны. В некоторых случаях, когда ленты разбиты на страницы или хранятся в архивах, может потребоваться выполнить дополнительные запросы, опять же с использованием гиперсреды, чтобы найти последнее обработанное событие. После извлечения необработанных событий производится их обработка по отдельности, в соответствии с предметными правилами, и изменяется идентификатор последнего обработанного события.

Возможно, вам будет интересно узнать, как поступать с ошибками, возникающими в ходе обработки событий. Решения интеграции на основе REST не имеют встроенной поддержки для обслуживания подозрительных или переходных сообщений, и эта проблема подробнее будет рассматриваться ближе к концу данной главы.

Чтобы завершить пример, нужно реализовать недостающую логику нижнего уровня, которая анализирует события из ленты. Она представлена в листингах 13.29 и 13.30. Также нужно добавить последнюю пару инструкций `using`:

```
using System.IO;
using System.Xml;
```

Листинг 13.29. Парсинг событий из ленты

```
private static BeganFollowing ParseEvent(SyndicationContent content)
{
    // ссылка на servicestack.text
    var jsonString = ParseFeedContent(content);
    var bf = JsonSerializer.DeserializeFromString<BeganFollowing>(jsonString);
    return bf;
}

private static string ParseFeedContent(SyndicationContent syndicationContent)
{
    using(var sw = new StringWriter())
    using(var xw = XmlWriter.Create(sw))
    {
        syndicationContent.WriteTo(xw, "BF", "BF");
        xw.Flush();

        return XDocument.Parse(sw.ToString()).Root.Value;
    }
}
```

Листинг 13.30. Модель предметного события в потребителе

```
public class BeganFollowing
{
    public string AccountId { get; set; }
    public string FollowerId { get; set; }
}
```


На этом разработка примера для данной главы завершена. Мы надеемся, что вы достаточно хорошо усвоили теорию и увидели достаточное количество практических примеров, чтобы чувствовать себя уверенно при рассмотрении REST как варианта интеграции ограниченных контекстов в своих проектах.

Напоследок осталось лишь проверить работу примера. Для этого можно попробовать добавить несколько новых последователей, как было показано выше. Следите при этом за окном консоли, которое появляется автоматически. Вы должны увидеть результаты своих действий, напоминающих изображенные на рис. 13.21. Можно также попробовать обратиться к ленте Atom непосредственно в браузере, чтобы убедиться в появлении в ней новых событий.

```
Processing event: pop_0 - 8989
Processing event: pop_0 - 56465
Processing event: 123 - 123
Processing event: 123 - 989
```

Рис. 13.21. Обработка событий потребителем

ПРИМЕЧАНИЕ

Если вам интересно узнать что-то еще или что-то осталось непонятным, вы можете обратиться со своими мыслями и вопросами на форум по адресу <http://p2p.wrox.com/>.

Сопровождение приложений REST

Так же как любая другая система, после первоначального развертывания приложение нуждается в поддержке и сопровождении. Под этим может подразумеваться поддержка совместимости новых версий прикладных интерфейсов, продолжающих развиваться, мониторинг качества работы системы или сбор показателей, используемых ответственными лицами при принятии решений.

Поддержка совместимости новых версий

Внося небольшие усовершенствования в API, легко можно избежать нарушения работоспособности существующих клиентов. Для этого нужно лишь гарантировать обратную совместимость. Представьте, что имеется приложение, в котором доступен ресурс «состояние доставки» (Shipping Status):

```
{
  "totalLegs": 5,
  "legsCompleted": 3,
  "currentVesselId": "sst399",
  "nextVesselId": "u223a"
}
```

и от вас потребовали добавить в него дополнительную информацию. Эта информация должна добавляться только в конец, как показано ниже:

```
{  
  "totalLegs": 5,  
  "legsCompleted": 3,  
  "currentVesselId": "sst399",  
  "nextVesselId": "u223a",  
  "eta": "2014-09-01"  
}
```

Это обратно совместимое изменение и рекомендуется потому, что оно не нарушит работу клиентов, привязанных к старому формату.

Перестройка API — более сложная задача. Она часто возникает, когда требуется внести в API крупные изменения, нарушающие обратную совместимость. Например, может понадобиться удалить ресурсы, перераспределить информацию между ресурсами или кардинально изменить форматы. В таких случаях обычно используют два основных варианта поддержки новых версий — включение номера версии в URI или в заголовок HTTP. Включение номера версии в URI обычно выражается в добавлении префикса, например: `/v2/accountmanagement/`. Другой вариант, с включением номера версии в заголовок, можно реализовать с использованием HTTP-заголовка `Version`, например такого: `Version: 2`.

Мониторинг и сбор показателей

Огромное преимущество использования HTTP заключается в существовании большого числа готовых инструментов мониторинга, которые можно просто подключить к своим API и тут же начать получать массу разнообразных показателей. Большой популярностью, например, пользуется инструмент New Relic (<http://newrelic.com/>), но он не бесплатный. Вместо стандартных или вместе с ними может понадобиться организовать сбор своих, специфических показателей. В подобных случаях часто используют такие инструменты, как StatsD (<https://github.com/etsy/statsd/>) и клиент StatsD для C# (<https://github.com/goncalopereira/statsd-csharp-client>).

Недостатки интеграции ограниченных контекстов с применением REST

Возможно, у вас уже начало складываться собственное мнение, но лишь одно можно сказать определенно: архитектура REST хорошо подходит для групп, желающих создавать масштабируемые и отказоустойчивые системы и старающихся избежать зависимости от фреймворков обмена сообщениями. Однако прежде чем решить, что REST — это правильный выбор, важно узнать о некоторых ее недостатках.

Одними из существенных недостатков REST в сравнении с системами обмена сообщениями являются большие трудозатраты и необходимость предварительного проектирования. Создание масштабируемых и отказоустойчивых систем на основе REST может потребовать приложения значительных усилий в самом начале. Существенная часть дополнительных трудозатрат объясняется необходимостью компенсировать отсутствие функциональных возможностей, уже реализованных

в решениях на основе обмена сообщениями. Но, ознакомившись со списком недостатков, имейте в виду, что по мере развития проекта они часто превращаются в преимущества. Вам потребуется поддерживать меньшее число фреймворков, и вы будете лучше понимать, как в действительности осуществляются взаимодействия в вашей распределенной системе.

ПРИМЕЧАНИЕ

Помните, что перечисленные здесь недостатки относятся в основном к использованию REST для создания асинхронных приложений, управляемых событиями. Они не всегда проявляются при использовании REST в других контекстах.

Меньшая отказоустойчивость

Применение решений, управляемых событиями на основе REST, повышает отказоустойчивость в сравнении с RPC, но не настолько, как решения на основе обмена сообщениями. В предыдущей главе мы достигли поставленной цели: разместить заказ и тут же сохранить его. Любые ошибки в процессе доставки команды `PlaceOrder` просто привели бы к повторной отправке сообщения. Но это не относится к системе на основе REST, созданной в данной главе. Если база данных `Event Store` окажется недоступна в момент попытки сохранить событие «Последовать за единомышленником» (`Began Following`), эта проблема не будет решена автоматически, когда `Event Store` опять окажется доступной.

Один из способов повысить отказоустойчивость — самостоятельно реализовать механизм с промежуточной буферизацией. Для этого может потребоваться добавить поддержку очередей там, где отказоустойчивость особенно важна для предприятия. С другой стороны, можно попробовать пойти путем обеспечения высокой доступности, добавив больше экземпляров приложения за балансировщиком нагрузки или создав кластер. `Event Store` поддерживает кластеризацию, поэтому это вполне жизнеспособный вариант для примера в данной главе.

Проще говоря, вам придется поработать, чтобы повысить отказоустойчивость до уровня, который фреймворки обмена сообщениями обеспечивают по умолчанию.

Потенциальная непротиворечивость

Слабосвязанные системы, не использующие общих ресурсов и взаимодействующие асинхронно, всегда склонны к потенциальной непротиворечивости. В эту категорию попадают также рекомендованные в этой главе REST-системы, управляемые событиями. Например, когда ограниченный контекст управления учетными записями экспортирует события «Последовать за единомышленником», он уже хранит их локально. Но потребители, опрашивающие ленту, не могут отреагировать на эти события немедленно, пока не извлекут ленту и не обработают их. Поэтому, в зависимости от используемых API, клиенты могут получить или не получить информацию, основанную на последних событиях.

Преодоление сложностей с потенциальной непротиворечивостью в решениях интеграции с применением REST основывается на тех же базовых принципах, что

и в системах обмена сообщениями. Вы должны отказаться от больших транзакций в пользу более мелких. Также следует двигаться вперед, к новым состояниям. Наконец, подумайте о возможности повторной отправки сообщений ограниченное число раз в надежде, что одна из попыток увенчается успехом.

Ключевые идеи

- Популярность протокола HTTP делает его серьезным кандидатом на применение для интеграции ограниченных контекстов.
- Кроме многих других преимуществ, HTTP ликвидирует зависимость от любых технологий и позволяет подбирать технологии по своему усмотрению.
- Использование HTTP подразумевает возможность использования одного и того же API для обслуживания внешних клиентов и для внутренних нужд.
- Протокол HTTP можно использовать множеством способов; его можно использовать для вызова удаленных процедур или для создания REST-систем, управляемых событиями.
- RPC может быть отличным выбором для простых решений, тогда как REST-системы, управляемые событиями, имеют лучшую масштабируемость и отказоустойчивость.
- В решениях RPC через HTTP можно использовать богатый возможностями, но чрезмерно подробный протокол SOAP или более легковесные форматы XML и JSON.
- Архитектура REST основывается на понятиях ресурсов, гиперсреды и отсутствии информации о состоянии, в точности как Всемирная паутина.
- REST позволяет использовать все преимущества соглашений и возможностей HTTP.
- В REST-системах, управляемых событиями, предметные события можно передавать как сообщения.
- Асинхронный опрос лент Atom, содержащих списки событий, образует фундамент для приложений на основе REST, управляемых событиями.
- В архитектуре REST все еще возможно использовать принципы SOA для создания слабосвязанных систем независимыми группами разработчиков.
- Запросы и ответы HTTP определяют контракт между ограниченными контекстами. Старайтесь избегать разрушительных изменений и стремитесь к поддержанию обратной совместимости, чтобы не нарушать нормальный рабочий ритм других групп.
- В каком бы виде вы ни использовали HTTP, всегда старайтесь сделать предметные понятия более явными.

ЧАСТЬ III

Тактические шаблоны: создание эффективных моделей предметной области

Глава 14: Знакомство со стандартными блоками моделирования предметной области

Глава 15: Объекты-значения

Глава 16: Сущности

Глава 17: Службы предметной области

Глава 18: События предметной области

Глава 19: Агрегаты

Глава 20: Фабрики

Глава 21: Репозитории

Глава 22: Регистрация событий

14

Знакомство со стандартными блоками моделирования предметной области

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Роль тактических шаблонов в создании эффективных объектно-ориентированных предметных моделей
- Знакомство с объектами-значениями, сущностями, предметными службами области и модулями, используемыми при моделировании предметной области и ее поведения
- Обзор шаблонов жизненного цикла: агрегат, фабрика и репозиторий
- Другие шаблоны предметных событий и их регистрации

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 14 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Отобразить программную модель обратно в аналитическую и гарантировать их тесную связь очень сложно. Чтобы помочь разработчикам и прояснить суть, Эванс взял за основу шаблон «Предметная модель», который впервые был классифицирован Мартином Фаулером в книге «Patterns of Enterprise Application Architecture». Он ввел язык шаблонов, содержащий множество стандартных заготовок для создания эффективных предметных моделей. Шаблоны, основанные на передовых объектно-ориентированных приемах, иногда называют тактическими шаблонами DDD. Многие шаблоны сами по себе не новы, но Эванс был первым, кто объединил их с целью помочь разработчикам в создании эффективных предметных моделей. Эта глава представляет собой обзорное знакомство с тактическими шаблонами предметно-ориентированного проектирования. Каждому шаблону будет посвящена отдельная глава в этой части книги, где мы рассмотрим их более подробно.

Несмотря на то что эта часть книги в целом и данная глава в частности подробно описывают приемы создания предметных моделей, тактика реализации этих моделей должна оставаться гибкой и открытой для инноваций. В своей книге Эванс одобрил объектно-ориентированный подход, однако это не повод забывать о других парадигмах проектирования, которые обсуждались в главе 5 «Шаблоны реализации предметной модели». Шаблон «Предметные события», с которым вы познакомитесь далее в этой главе, первоначально не был включен в число стандартных шаблонов, о чем Эванс впоследствии высказал свое сожаление. Кроме того, все большую популярность для выражения предметных моделей начинают приобретать функциональное программирование и прием регистрации событий (см. главу 18 «События предметной области»).

Шаблоны, используемые для создания предметных моделей и связывания программной модели с аналитической, непрерывно развивались с момента выхода книги Эванса. Семантика создания предметных моделей может и будет изменяться, главное, чтобы для представления идей и понятий в программном коде использовался язык предметной области — единый язык.

Тактические шаблоны

Роль тактических шаблонов в DDD состоит в том, чтобы управлять сложностью и гарантировать ясность предметной модели. Они помогают отразить и передать назначение, закономерности и логику предметной области. Шаблоны строятся на прочной основе объектно-ориентированных принципов, многие из которых описаны в широко известных книгах по проектированию, а именно «Patterns of Enterprise Application Architecture» Мартина Фаулера и «Design Patterns: Elements of Reusable Object-Oriented Software» Ральфа Джонсона, Джона Влиссидеса, Ричарда Хелма и Эриха Гамма¹.

Каждый стандартный шаблон отвечает за что-либо одно; они могут представлять предметные понятия, например сущность (entity) и объект-значение (value object), или гарантировать защиту от перенасыщения предметных понятий деталями, имеющими отношение к жизненному циклу, например фабрика и репозиторий. В некотором смысле стандартные шаблоны можно рассматривать как своеобразный единый язык для разработчиков, основу для конструирования выразительных и эффективных предметных моделей.

Для создания предметной модели можно использовать множество стандартных шаблонов, как показано на рис. 14.1. Но обратите внимание, что шаблон «Прикладные службы» относится к клиенту предметной модели и потому не будет рассматриваться в этой части книги. Прикладные службы обсуждаются в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования».

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001. — *Примеч. пер.*

Шаблоны моделирования предметной области

Следующие шаблоны представляют правила и логику, действующие в предметной области. Они выражают отношения между объектами, моделируют правила и связывают аналитическую модель с программной. Эти шаблоны отражают элементы модели в программном коде.

Сущности

Сущность (entity) представляет предметное понятие, которое определяется его индивидуальностью, а не атрибутами. Индивидуальность объекта, в отличие от его атрибутов, не меняется на протяжении его жизненного цикла. Ответственность сущности заключается в определении того, что значит быть тем же самым; в коде это часто достигается путем переопределения операций равенства класса.

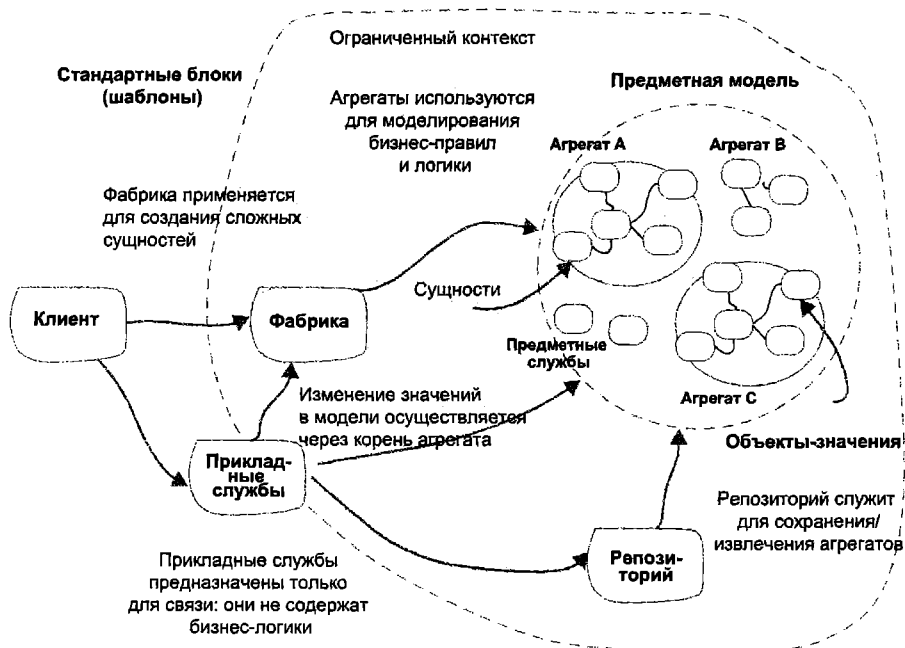


Рис. 14.1. Тактические шаблоны — стандартные блоки предметной модели

Примером сущности может служить продукт; его уникальная индивидуальность не изменяется, но описание, цена и прочие атрибуты могут изменяться много раз. Сущности являются изменчивыми объектами, так как их атрибуты могут изменяться.

На рис. 14.2 изображены основные понятия сущности.

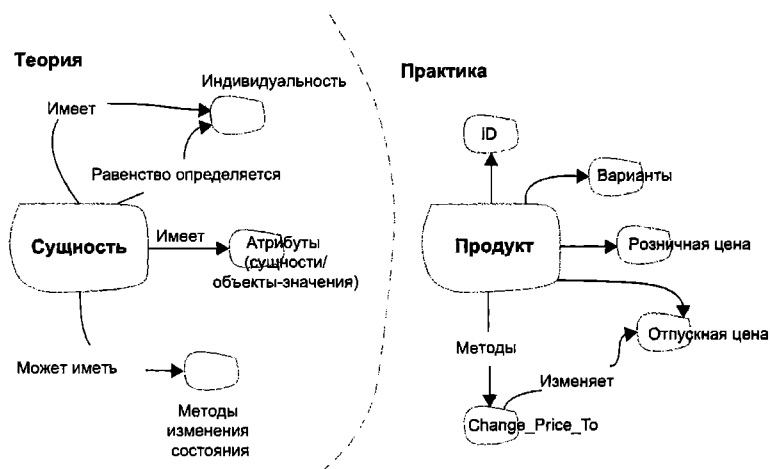


Рис. 14.2. Сущность

В листинге 14.1 демонстрируется реализация продукта в виде сущности.

Листинг 14.1. Сущность Product

```
public class Product : Entity<Guid>
{
    private readonly IList<Option> _options;
    private Price _selling_price;
    private Price _retail_price;

    public Product(Guid id, Price sellingPrice, Price retailPrice)
        : base(id)
    {
        _selling_price = sellingPrice;
        if (!selling_price_matches(retailPrice))
            throw new PricesNotInTheSameCurrencyException(
                "Selling and retail price must be in the same currency");
        _retail_price = retailPrice;
        _options = new List<Option>();
    }

    public void change_price_to(Price new_price)
    {
        if (!selling_price_matches(new_price))
            throw new PricesNotInTheSameCurrencyException(
                "You cannot change the price of this
                product to a different currency");
        _selling_price = new_price;
    }

    public Price savings()
    {

```

```

    Price savings = _retail_price.minus(_selling_price);

    if (savings.is_greater_than_zero())
        return savings;
    else
        return new Price(0m, _selling_price.currency);
}

private bool selling_price_matches(Price retail_price)
{
    return _selling_price.Equals(retail_price);
}

public void add(Option option)
{
    if (!this.contains(option))
        _options.Add(option);
    else
        throw new ProductOptionAddedNotUniqueException(
            string.Format(
                "This product already has the option {0}",
                option.ToString()));
}

public bool contains(Option option)
{
    return _options.Contains(option);
}
}

```

В листинге 14.1 видно, что идентификатор продукта устанавливается при его создании, а методы для его изменения отсутствуют. Сущность **Product** делегирует выполнение всей работы объектам-значениям **Money** и **Option** — атрибутам/характеристикам объекта **Order**. Сущность **Product** инкапсулирует данные и экспортирует поведение (операции); данные класса скрыты и недоступны извне.

Также обратите внимание, что сущность **Product** наследует обобщенный базовый класс, который параметризуется типом, используемым для идентификации. В базовом классе имеются доступные для переопределения методы сравнения, подобные тем, что вы увидите в объекте-значении **Money**. Но в отличие от последнего, сравнение объектов выполняется по их типам и идентификаторам.

Для полноты изложения в листинге 14.2 приводится реализация базового класса **Entity**.

Листинг 14.2. Базовый класс **Entity**

```

public abstract class Entity<TId> : IEquatable<Entity<TId>>
{
    private readonly TId id;

    protected Entity(TId id)
    {

```

```
        if (object.Equals(id, default(TId)))
        {
            throw new ArgumentException(
                "The ID cannot be the default value.", "id");
        }

        this.id = id;
    }

    public TId Id
    {
        get { return this.id; }
    }

    public override bool Equals(object obj)
    {
        var entity = obj as Entity<TId>;
        if (entity != null)
        {
            return this.Equals(entity);
        }
        return base.Equals(obj);
    }

    public override int GetHashCode()
    {
        return this.Id.GetHashCode();
    }

    public bool Equals(Entity<TId> other)
    {
        if (other == null)
        {
            return false;
        }
        return this.Id.Equals(other.Id);
    }
}
```

Как видно из листинга 14.2, наследуя этот базовый класс, вы сохраняете логику, определяющую понятие равенства сущностей, внутри самой сущности. Абстрактный базовый класс берет на себя все хлопоты, связанные с индивидуальностью и проверкой на равенство, чтобы при реализации своего класса вы могли сосредоточиться на предметной логике.

Объекты-значения

Объекты-значения (value objects) представляют элементы или понятия предметной области, известные только по их характеристикам; используются как описатели элементов в модели; не обладают уникальной индивидуальностью. Из-за того что объекты-значения не обладают индивидуальностью в рамках модели,

они определяются атрибутами, то есть атрибуты определяют их индивидуальность. Объекты-значения не нуждаются в индивидуальности, потому что всегда связаны с другими объектами и их назначение очевидно из конкретного контекста. Например, в модели может иметься сущность, представляющая заказ, которая использует объекты-значения для хранения адреса доставки, элементов заказа, информации о курьере и т. д. Ни одна из этих характеристик не обладает индивидуальностью сама по себе, потому что имеет смысл только в контексте заказа. Адрес доставки без самого заказа не имеет смысла. Объекты-значения сравниваются между собой по значениям их атрибутов и, так же как сущности, отвечают за реализацию операций сравнения.

Так как объекты-значения определяются атрибутами, они считаются неизменяемыми, то есть после создания они никогда не изменяют свое состояние. Отличным примером объекта-значения могут служить денежные средства. Неважно, что вы не сможете отличить друг от друга пять однофунтовых монет или однодолларовых банкнот, лежащих в вашем кармане. Вас наверняка не беспокоит индивидуальность валюты — значение имеет только сумма. Если кто-то заменит пятидолларовую банкноту в вашем бумажнике пятью однодолларовыми банкнотами, это не изменит тот факт, что у вас есть пять долларов. Разумеется, в действительности банкноты имеют уникальные идентификаторы в форме серий и номеров, но предметная модель не является отражением действительности. Это — абстракция, сконструированная для удовлетворения потребностей бизнес-сценариев использования. На рис. 14.3 изображены основные понятия объекта-значения.

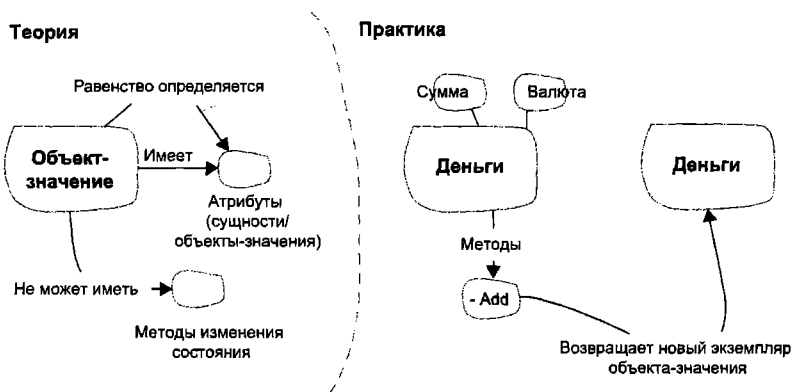


Рис. 14.3. Объект-значение

Листинг 14.3 представляет модель денежных средств в виде объекта-значения.

Листинг 14.3. Объект-значение, представляющий денежные средства

```
public class Money
{
    protected readonly decimal _value;
    private readonly Currency _currency;

    public Money()
```

```
        : this(0m, new SterlingCurrency())
    {
    }

    public Money(decimal value, Currency currency)
    {
        _value = value;
        _currency = currency;
    }

    public Money add(Money to_add)
    {
        if (this._currency.Equals(to_add._currency))
            return new Money(_value + to_add._value, _currency);
        else
            throw new NonMatchingCurrencyException(
                "You cannot add money with different currencies.");
    }

    public Money minus(Money to_discount)
    {
        if (this._currency.Equals(to_discount._currency))
            return new Money(_value - to_discount._value, _currency);
        else
            throw new NonMatchingCurrencyException(
                "You cannot remove money with different currencies.");
    }

    public override string ToString()
    {
        return string.Format("{0}",
            _currency.format_for_displaying(_value));
    }

    // Переопределение методов сравнения

    public static bool operator ==(Money money, Money money_to_compare)
    {
        if ((object)money == null)
        {
            return (object)money_to_compare == null;
        }
        return money._value == money_to_compare._value &&
            money._currency == money_to_compare._currency;
    }

    public static bool operator !=(Money money, Money money_to_compare)
    {
        return (money._value != money_to_compare._value ||
            money._currency != money_to_compare._currency);
    }

    public bool Equals(Money obj)
    {

```

```

        if (obj == null) return false;
        return obj._value == _value && obj._currency.equals(_currency);
    }

    public override bool Equals(System.Object obj)
    {
        if (obj == null) return false;
        if (obj.GetType() != this.GetType()) return false;

        return ((Money)obj)._value == _value &&
            obj._currency.equals(_currency);
    }

    public override int GetHashCode()
    {
        return (_value.GetHashCode() + _currency.GetHashCode())
            .GetHashCode();
    }
}

```

Как можно заметить в листинге 14.3, здесь переопределяются методы сравнения, то есть объекты `Money` сравниваются только по их атрибутам; в данном случае атрибутами являются сумма (`_value`) и валюта (`_currency`). Класс определяет неизменяемые объекты. После создания объекта нельзя изменить его состояние. Метод `add` возвращает новый объект `Money`. Это называется замыканием операций, потому что состояние первоначального объекта не изменяется. Для представления валюты используется еще один объект-значение. Сравнение валют делегируется этому объекту, который также должен переопределить методы сравнения.

Предметные службы

Предметные службы содержат предметную логику и понятия, которые нельзя смоделировать в виде объектов-значений или сущностей. Предметные службы не имеют индивидуальности или состояния, они отвечают за управление бизнес-логикой использования сущностей и объектов-значений. Отличным примером предметной службы является калькулятор стоимости доставки, представленный в листинге 14.4. Эта служба является бизнес-функцией, которая на основе коллекции грузов (объектов-значений) и весов пакетов может вычислить стоимость доставки. Данную функциональность сложно оформить в виде объекта предметной области, поэтому лучше представить ее в виде предметной службы.

Листинг 14.4. Предметная служба «Калькулятор стоимости доставки»

```

public class ShippingCostCalculator
{
    private IEnumerable<WeightBand> _weightBand;
    private readonly WeightInKg _boxWeightInKg;

    public ShippingCostCalculator(IEnumerable<WeightBand> weightBand,
        WeightInKg boxWeightInKg)
    {
        _weightBand = weightBand;
        _boxWeightInKg = boxWeightInKg;
    }
}

```

```
{
    _weightBand = weightBand;
    _boxWeightInKg = boxWeightInKg;
}

public Money CalculateCostToShip(IEnumerable<Consignment> consignments)
{
    var weight = GetTotalWeight(consignments);

    // Список, отсортированный в обратном порядке
    _weightBand = _weightBand.OrderBy(x =>
        x.ForConsignmentsUpToThisWeightInKg.Value);

    // Получить первое совпадение
    var weightBand = _weightBand.FirstOrDefault(x =>
        x.IsWithinBand(weight));

    return weightBand.Price;
}

private WeightInKg GetTotalWeight(
    IEnumerable<Consignment> consignments)
{
    var totalWeight = new WeightInKg(0m);

    // Вычислить вес элементов
    foreach (Consignment consignment in consignments)
        totalWeight = totalWeight.Add(consignment.ConsignmentWeight());

    // Добавить вес упаковки
    totalWeight = totalWeight.Add(_boxWeightInKg);

    return totalWeight;
}
}
```

Класс `ShippingCostCalculator` содержит предметную логику вычисления стоимости доставки груза исходя из веса коллекции товаров и веса упаковки. Организовав логику в виде предметной службы и дав ей имя, вы сможете явно сообщать специалистам в предметной области о конкретном элементе предметной логики, таком как правило или процесс, используя более краткую терминологию. Возможно, калькулятор `ShippingCostCalculator` прежде был неявным понятием на предприятии, но, дав ему имя, вы сделали его явным, и теперь оно должно быть добавлено в единый язык и в аналитическую модель.

Модули

Модули в C# реализуются в виде пространств имен или проектов. Они используются для организации и объединения родственных понятий (сущностей и объектов-значений), чтобы упростить представление больших предметных моделей.

Модули используются для разложения предметных моделей на составные части. Не путайте их с подобластями, которые используются для разложения предметной области и ограниченных контекстов с целью ограничить область применения предметной модели. Как показано на рис. 14.4, имена модулей берутся прямо из единого языка и помогают отличать части предметной модели, которые должны рассматриваться независимо друг от друга. Модули позволяют разработчикам быстро прочитать и понять предметную модель в коде, прежде чем углубиться в изучение файлов с классами. Они также действуют как границы ответственности, ясно очерчивающие части предметной модели и гарантирующие минимальное число связей между предметными объектами. Применяйте модули для обеспечения слабой связанности и высокой согласованности внутри предметной модели. Старайтесь ограничивать внутреннее содержимое модуля только тесно связанным множеством функций и особенностей.

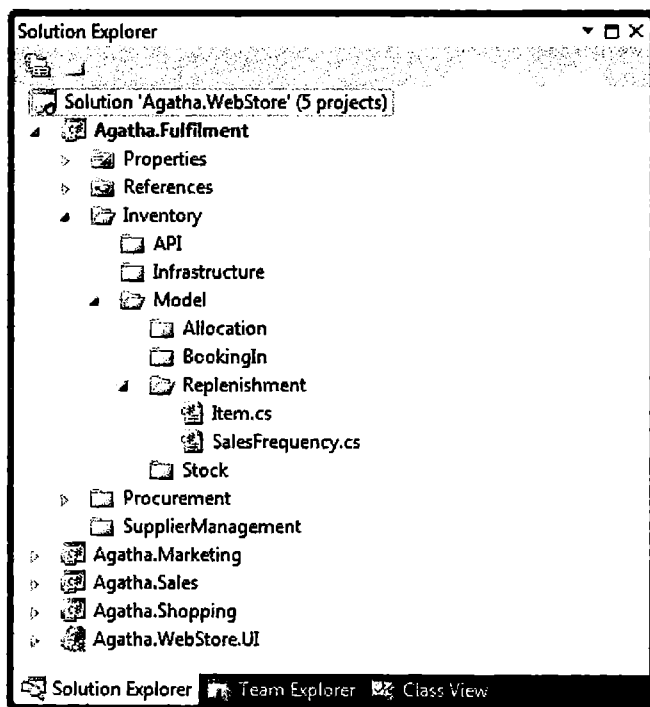


Рис. 14.4. Модули используются для организации предметных понятий внутри предметной области

Шаблоны жизненного цикла

Далее перечислены шаблоны, имеющие отношение к созданию и хранению объектов, представляющих структуру предметной области.

Агрегаты

Сущности и объекты-значения осуществляют взаимодействие друг с другом для образования сложных отношений, соответствующих инвариантам внутри предметной модели. При работе с большими комплексами объектов, обладающих многочисленными связями между собой, часто трудно гарантировать непротиворечивость и параллельность операций с предметными объектами. На рис. 14.5 изображен большой граф объектов. Попытка интерпретировать эту коллекцию объектов как одно целое может оказаться весьма непростым делом и привести к появлению проблем с производительностью в приложении. Например, было бы нежелательно запрещать клиенту изменять адрес в его учетной записи только потому, что параллельно изменяется состояние его заказа. Эти две характеристики не связаны между собой и не имеют общих границ в смысле непротиворечивости или параллелизма.

Предметно-ориентированное проектирование имеет шаблон «Агрегат» (Aggregate), помогающий поддерживать согласованность и определяющий границы параллельных транзакций для графов объектов. Большие модели разбиваются на инварианты и группируются в агрегаты сущностей и объектов-значений, которые интерпретируются как одно целое. Как показано на рис. 14.6, модель можно дистриллировать в агрегаты.

Отношения между корневыми агрегатами должны реализовываться с применением ссылки на идентификатор (ID) другого агрегата, а не на сам объект, как показано на рис. 14.7. Следование этому принципу поможет сохранить неприкосновенность границ между агрегатами и избежать необходимости загружать большие графы объектов, значительная часть из которых может не потребоваться.

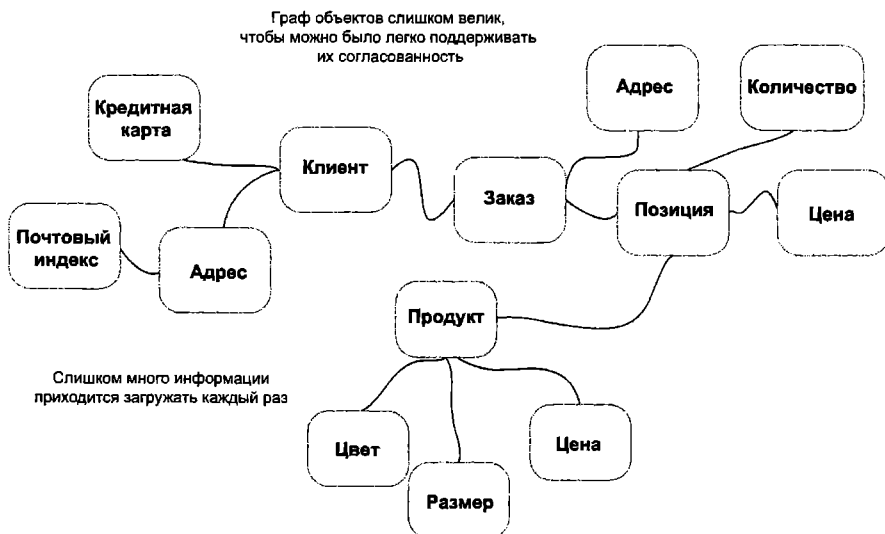


Рис. 14.5. Большой граф объектов

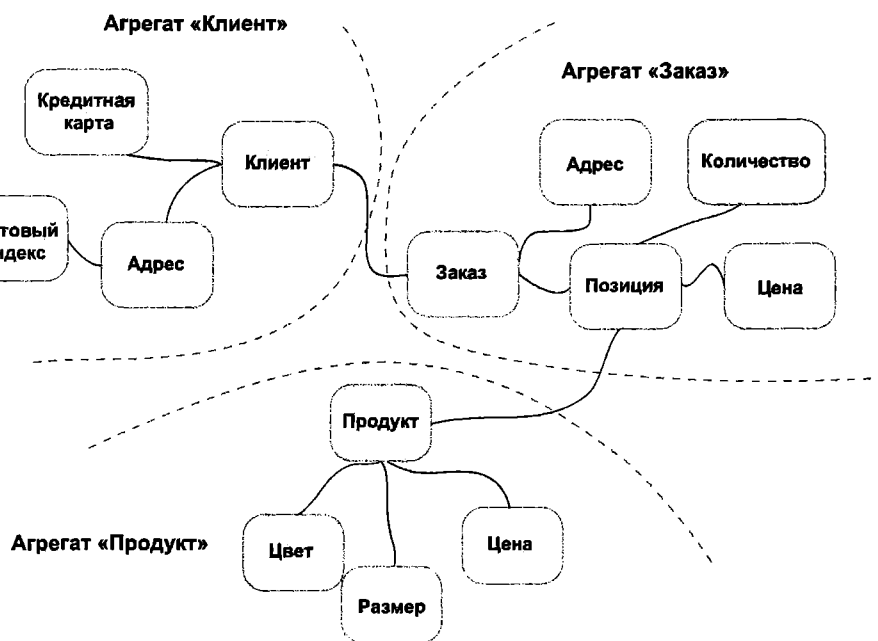


Рис. 14.6. Большой граф объектов, разбитый на агрегаты

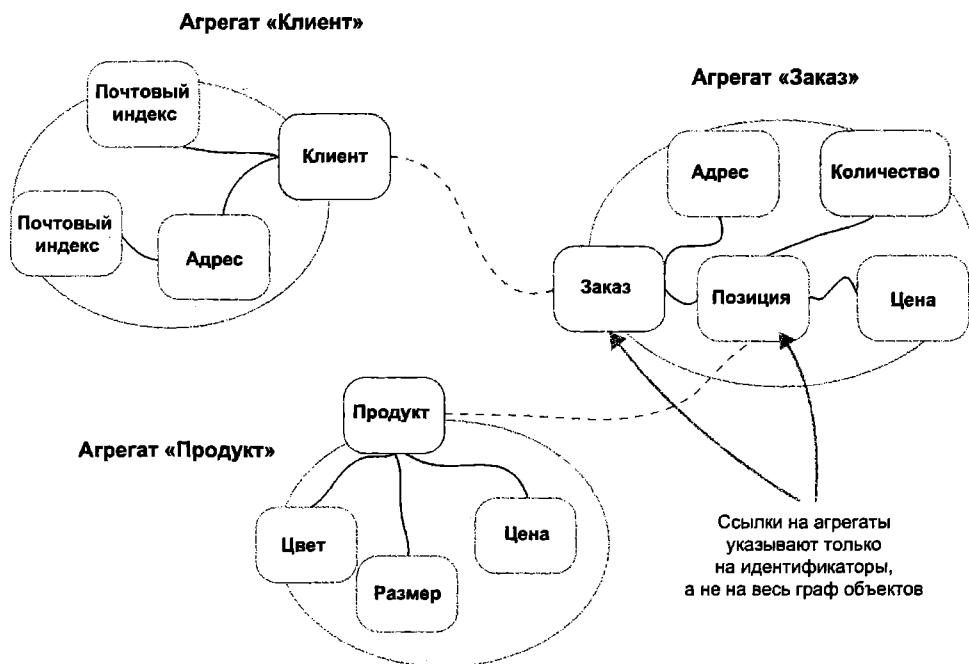


Рис. 14.7. Корень агрегата действует как точка входа

Группировка агрегатов, изображенная на рис. 14.6 и 14.7, на первый взгляд кажется вполне разумным способом разделения графа объектов; однако определение групп агрегатов исключительно на родственных понятиях дает лишь возможность ослабить проблемы, связанные с поддержанием целостности и параллельным доступом. Возьмем для примера агрегат «Клиент». Если потребуется исправить адрес в то же время, когда изменяются некоторые персональные данные клиента, может возникнуть проблема блокировки. Хотя эти два понятия и связаны с клиентом, но к ним не предъявляются требования инвариантности. Поэтому их можно разбить на два отдельных агрегата и использовать ту же логику обработки кредитных карт, а затем сгруппировать все агрегаты в модуле «Клиент», как показано на рис. 14.8.

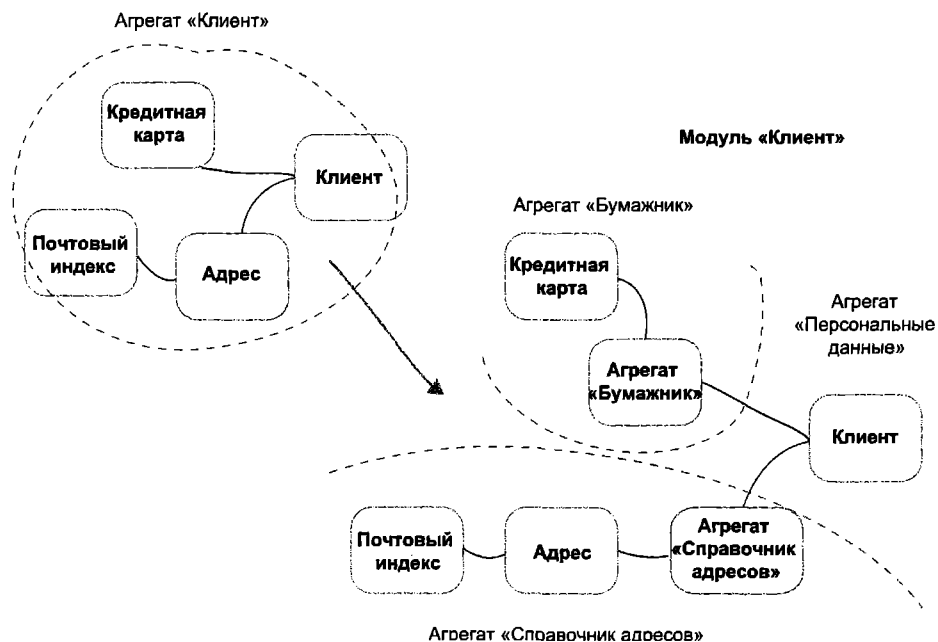


Рис. 14.8. Агрегаты должны определяться на основе инвариантов

ЧТО ТАКОЕ ИНВАРИАНТ?

Инварианты (invariants) — это правила, обеспечивающие согласованность в предметной модели. Любые изменения в сущностях или агрегатах выполняются в соответствии с бизнес-правилами. Примером инварианта может служить правило, требующее, чтобы клиент всегда имел полный адрес. Для соблюдения этого инварианта достаточно не дать пользователю возможности править отдельные строки в адресе, из-за чего адрес может стать недействительным. Интерпретируйте его, например, как объект-значение, чтобы гарантировать, что получившийся в результате адрес останется действительным.

Корень агрегата, изображенный на рис. 14.9, действует как точка входа. Никакие другие сущности или объекты-значения за пределами агрегата не могут получить ссылку на объект внутри агрегата. Объекты за пределами агрегата могут ссылаться только на корень этого агрегата. Любые изменения в объектах, принадлежащих агрегату, должны осуществляться только через его корень. Корень скрывает данные агрегата и экспортирует только методы для их изменения.

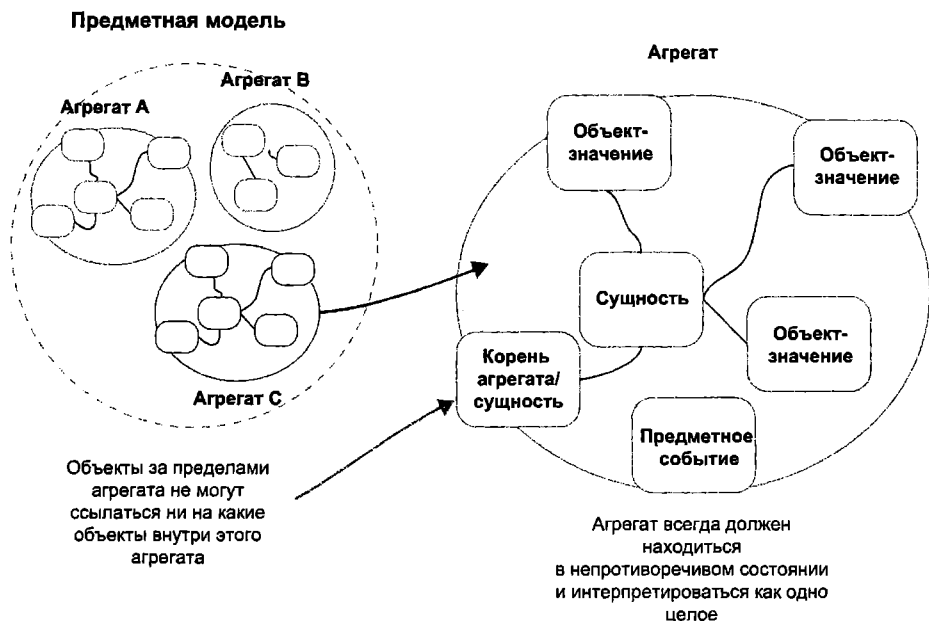


Рис. 14.9. Корень агрегата действует как точка входа

Фабрики

Если создание сущности или объекта-значения связано с выполнением особенно сложных процедур, их конструирование должно быть возложено на фабрики (factory), как показано на рис. 14.10. Фабрика гарантирует соблюдение всех инвариантов прежде, чем предметный объект будет создан. Если предметный объект прост и для создания действительного объекта не требуется соблюдать какие-то

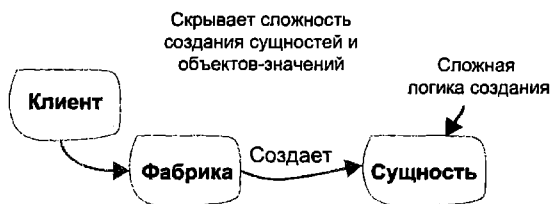


Рис. 14.10. Фабрика

особые правила, предпочтение следует отдавать методу-конструктору. Фабрики можно также использовать для воссоздания предметных объектов на основе информации из постоянного хранилища.

Из листинга 14.5 видно, что сущность `Customer` имеет фабричный метод, обеспечивающий создание адреса с допустимым идентификатором клиента.

Листинг 14.5. Фабричный метод сущности

```
public class Customer : Entity<Guid>
{
    public Customer(Guid id)
        : base(id)
    {
    }
    // ...
    public Address create(Address delivery_address)
    {
        return new Address(Guid.NewGuid(), this.Id);
    }
}
```

Репозитории

Предметной модели нужны методы для сохранения и восстановления агрегатов. Агрегат интерпретируется как единое целое, поэтому не должно быть никакой иной возможности сохранения изменений, кроме как через сохранение всего агрегата. Репозиторий (или хранилище), как показано на рис. 14.11, — это шаблон,

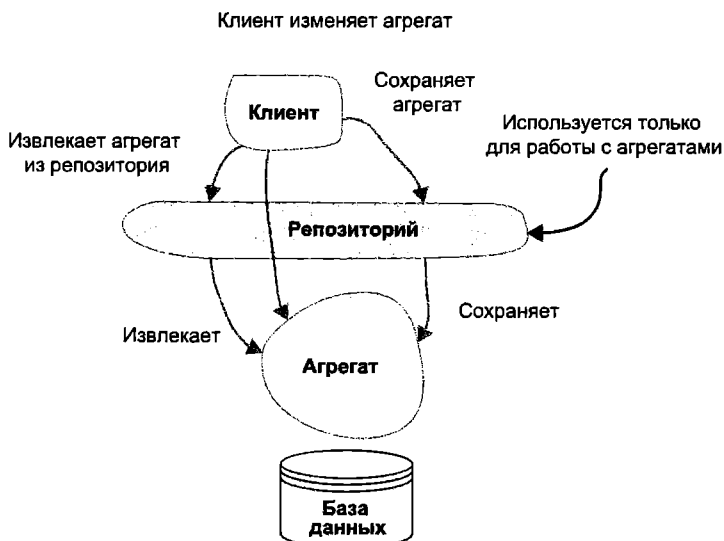


Рис. 14.11. Репозиторий

отделяющий хранилище данных от модели и позволяющий создавать модели, не отвлекаясь на решение инфраструктурных задач. Репозиторий — это механизм сохранения и извлечения агрегатов. Для отображения данных репозиторий не требуется, а для составления отчетов наиболее эффективным методом является выполнение прямых запросов к хранилищу данных. Репозиторий — это инфраструктурная задача, поэтому ее не всегда нужно отделять от фреймворка, лежащего в основе и выполняющего всю рутинную работу. Возможно, более целесообразно использовать в качестве репозитория фреймворки объектно-реляционного отображения (Object Relational Mapper, ORM), примерами которых могут служить NHibernate, RavenDB и Entity Framework. Многие разработчики уделяют этому шаблону слишком много внимания. Рассматривайте его лишь как способ сохранения и восстановления данных. Воспринимайте репозиторий как элемент инфраструктуры и не стремитесь абстрагировать его.

Другие шаблоны

С момента выхода книги Эрика Эванса появились еще два шаблона, которые могут пригодиться при создании предметных моделей, а именно шаблон «Предметные события» (domain events), о котором подробно рассказывается в главе 18, и шаблон «Регистрация событий» (event sourcing), описываемый в главе 22 «Регистрация событий».

Предметные события

Предметные события информируют о чем-то, случившемся в предметной области и важном для предприятия. События можно использовать для записи изменений в модели с целью последующего изучения или для взаимодействий между агрегатами. Часто операции с корнем агрегата могут приводить к побочным эффектам за его пределами. Другие агрегаты внутри модели также могут следить за событиями и действовать соответственно.

Например, рассмотрим корзину покупателя на сайте электронного магазина, изображенную на рис. 14.12. Каждый раз, когда клиент добавляет новый товар в корзину, желательно обновлять список рекомендуемых товаров, отображаемых на сайте. В этом случае после появления предметного события с информацией о содержимом корзины выполняется обновление списка рекомендаций для клиента. На получение события подписан ограниченный контекст рекомендаций. Без использования предметного события пришлось бы явно связать ограниченный контекст корзины с контекстом рекомендаций. Предметные события обеспечивают более естественный способ взаимодействий, с упором на конкретные моменты времени.

В листинге 14.6 показано определение события, публикуемого корзиной.

В листинге 14.7 демонстрируется служба выдачи рекомендаций, обрабатывающая события, генерируемые ограниченным контекстом корзины.

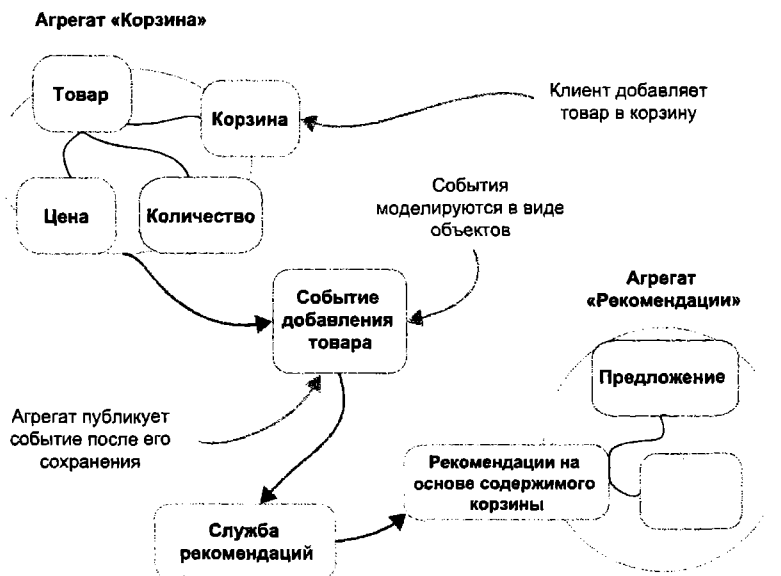


Рис. 14.12. Предметное событие

Листинг 14.6. Предметные события, генерируемые сущностью Basket

```

namespace Domain.Shopping
{
    public class Basket
    {
        private BasketItems _items;
        private Guid _id;

        // ....

        public void add(Product product)
        {
            _items.Add(BasketItemFactory.CreateItemFor(product, this));

            DomainEvents.raise(new ProductAddedToBasket(this._id,
                _items.ids_of_items_in_basket()));
        }
    }

    public class ProductAddedToBasket
    {
        public Guid basket_id { get; private set; }
        public IEnumerable<Guid> items_in_basket { get; private set; }

        public ProductAddedToBasket(Guid basket_id,
            IEnumerable<Guid> items_in_basket)
    }
}

```

```
        {  
            basket_id = basket_id;  
            items_in_basket = items_in_basket;  
        }  
    }  
    // .....  
}
```

Листинг 14.7. Обработчик предметных событий

```
namespace Domain.Recommendations  
{  
    public class RecommendationsEventHandlers  
    {  
        // .....  
  
        public void Handle(Shopping.ProductAddedToBasket product_added)  
        {  
            _recommendation_service.update_suggestions_for(  
                continues  
                product_added.basket_id, product_added.items_in_basket);  
        }  
    }  
}
```

На данном этапе не следует обращать пристальное внимание на конкретный синтаксис, используемый в данной главе; он подробно будет обсуждаться в главе 18. Предметные события играют важную роль в реализации.

Регистрация событий

Популярной альтернативой традиционному приему сохранения мгновенных снимков является регистрация событий (event sourcing). Вместо сохранения в базе данных состояния сущности при использовании приема регистрации событий сохраняются события, которые привели к определенному состоянию. Сохранение всех событий расширяет аналитические возможности приложения. Вместо текущего состояния сущности появляется возможность узнать, каким было это состояние некоторое время тому назад, как показано на рис. 14.13.

Возможность узнать состояние предметной модели в любой момент в прошлом дает большое конкурентное преимущество, потому что позволяет сопоставить события, происходившие в реальном мире, с изменениями в состоянии предметной модели. Для туристических агентств с представительствами в сети Интернет может быть интересно, почему в каком-либо месяце произошло существенное падение числа заказов. Благодаря регистрации события они смогут воссоздать состояние своего каталога туров и воспроизвести операции поиска, выполнявшиеся клиентами, чтобы понять, почему они не нашли то, что искали.

Как практик DDD, вы должны не только предложить предприятию возможность использовать выгоды от регистрации событий, но также искать новые пути моде-



Рис. 14.13. Сохранение последовательности событий вместо мгновенного снимка

лирования предметной области, обеспечивающие лучшую поддержку этого приема. В частности, вы должны стараться уйти от сохранения дампов с состоянием сущностей в базе данных с применением ORM. А вместо этого искать способы моделирования, перехвата и сохранения предметных событий. Вам может даже потребоваться добавить функции сохранения событий в существующую базу данных. Все эти задачи подробно рассматриваются в главе 22.

Ключевые идеи

- Тактические шаблоны DDD — это стандартные шаблоны, созданные Эвансом и основанные на шаблонах Мартина Фаулера, которые предназначены для использования в объектно-ориентированных предметных моделях.
- Стандартные шаблоны служат руководством для создания эффективных моделей, но это всего лишь руководство. Конкретные реализации предметных моделей могут значительно отличаться друг от друга, поэтому старайтесь не заикливаться на стандартных шаблонах.
- Сущности:
 - Определяются их индивидуальностью.
 - Индивидуальность не изменяется с течением времени.
 - Несут ответственность за проверку на равенство.
- Объекты-значения:
 - Описывают свойства и характеристики в пределах предметной области.
 - Не имеют индивидуальности.

- Являются неизменяемыми объектами в том смысле, что их состояние не может изменяться. Вместо этого объекты-значения, моделирующие свой-ства, должны заменяться новыми объектами.

○ Предметные службы:

- Содержат предметную логику, которую невозможно естественным спосо-бом реализовать в виде сущности или объекта-значения. В отличие от них прикладные службы лишь организуют выполнение предметной логики, но не реализуют ее.
- Не имеют внутреннего состояния, поэтому их можно вызывать многократ-но с одними и теми же исходными данными, и они всегда будут возвращать один и тот же результат.

○ Модули:

- Используются для декомпозиции, организации и увеличения удобочитае-мости предметной модели.
- Пространства имен в реализациях модулей можно применять для ослабле-ния внешних и усиления внутренних связей в предметной модели.
- Дают читателям возможность быстро понять организацию предметной мо-дели.
- Помогают ясно определить границы между предметными объектами.
- Инкапсулируют понятия, которые можно рассматривать независимо друг от друга. Находятся на более высоком уровне абстракции, нежели агрегаты и сущности.

○ Агрегаты:

- Позволяют разложить большие графы объектов на более мелкие группы предметных объектов с целью уменьшить сложность технической реализа-ции предметной модели.
- Представляют предметные понятия, а не только обобщенные коллекции предметных объектов.
- Основываются на предметных инвариантах.
- Определяют границы согласования, чтобы гарантировать непротиворечи-вость состояния предметной модели.
- Гарантируют правильную расстановку границ параллельных транзакций, чтобы обеспечить более высокую эффективность приложения за счет ис-ключения возможности блокировок на уровне базы данных.

○ Фабрики:

- Используются отдельно от конструирования.
- Инкапсулируют сложную логику создания сущностей и объектов-зна-чений.

○ Репозитории:

- Экспортируют интерфейс для доступа к коллекциям корней агрегатов в памяти.
- Не должны использоваться для создания отчетов.
- Предназначены для сохранения и извлечения корней агрегатов.
- Ослабляют связь предметного уровня с инфраструктурным кодом и со стратегиями работы с базами данных.

○ Предметные события:

- Важнейшие происшествия в действительной предметной области, представляющие особый интерес для сотрудников предприятия; являются частью единого языка.
- Дополнительный шаблон проектирования, помогающий сделать предметные события более явными в программном коде.
- Действуют подобно приемам публикации/подписки, где события сначала возбуждаются и затем обрабатываются обработчиками.

○ Регистрация событий:

- Заменяет традиционное сохранение мгновенных снимков состояния полной историей событий, приведших к текущему состоянию.
- Открывает широкие возможности для получения состояния в некоторые моменты времени в прошлом, известные как временные запросы (temporal queries).

15

Объекты-значения

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в объекты-значения, компоненты моделей в DDD
- Что такое объекты-значения и когда они используются
- Шаблоны для работы с объектами-значениями
- Средства, доступные для сохранения объектов-значений в базах данных NoSQL и SQL

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 15 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Предметная модель содержит сущности, которые сродни героям фильмов. И так же как герои фильмов, сущности часто обладают характеристиками, которые делают их интересными или полезными. Кто-то мог бы охарактеризовать Джеймса Бонда как весьма обаятельного человека. Аналогично кто-то мог бы посчитать интересной сущность `BankAccount`, имеющую атрибут `Balance` с кругленькой суммой. Моделирование этих важных описательных характеристик возлагается на компонент DDD под названием *объект-значение* (value object).

Объекты-значения не имеют индивидуальности. Они служат исключительно для описания атрибутов сущностей, имеющих отношение к предметной области и обычно выраженных в количественной форме. Отсутствие индивидуальности делает работу с объектами-значениями относительно простой и приятной. В частности, двумя основными характеристиками, обеспечивающими простоту использования объектов-значений, являются их неизменяемость и способность комбинироваться. В этой главе вы поближе познакомитесь с этими и другими их характеристиками. Также вы увидите некоторые распространенные шаблоны

моделирования объектов-значений, обеспечивающие удобство их использования и выразительность.

Применение объектов-значений иногда может вызывать сложности из-за обстоятельств, требующих глубокого осмысления, к которым относятся, в частности, сохранение, валидация и элементарные исключения. Все эти проблемы тоже будут рассматриваться в данной главе, чтобы вы обрели все необходимые знания для создания собственных объектов-значений.

Когда использовать объекты-значения

Объекты-значения определяют состояние сущности, описывая некоторые ее параметры или то, чем они владеют. Судно может иметь максимальную грузоподъемность, продуктовый магазин может иметь уровень запасов, а финансовый отчет может показывать квартальный товарооборот. Каждое из этих отношений можно смоделировать как отношение между сущностью и объектом-значением. Обратите внимание, что в каждом из этих примеров объект-значение представляет определенное понятие, имеющее единицу измерения и величину, или значение, то есть объект значения. Эти две ключевые характеристики делают объекты-значения важными техническими конструкциями в DDD.

Представление описательного понятия, не имеющего индивидуальности

При целенаправленном рассмотрении объектов-значений, таких как `Money` и другие, демонстрируемых повсюду в этой главе, можно заметить их логическую общность. У всех у них отсутствует понятие индивидуальности. Только сущности наделены индивидуальностью. Простым примером может служить банковский счет; вам почти наверняка потребуется реализовать поиск сущности `BankAccount` по ее идентификатору (ID), но понадобится ли искать баланс по его идентификатору? В большинстве случаев — нет, потому что баланс сам по себе не имеет никакого смысла.

В листинге 15.1 приводится реализация сущности `BankAccount`, баланс в которой представлен объектом `Money`. После вышесказанного вам должно быть понятно, почему такая реализация является наиболее разумной.

Листинг 15.1. Использование объекта-значения для представления понятия, не имеющего индивидуальности

```
public class BankAccount
{
    public BankAccount(Guid id, Money startingBalance)
    {
        this.Id = id;
        this.Balance = startingBalance;
    }
}
```

```
public Guid Id { get; private set; }

public Money Balance { get; private set; }

...
}

// объект-значение
public class Money
{
    public Money(int amount, Currency currency)
    {
        this.Amount = amount;
        this.Currency = currency;
    }

    private int Amount { get; set; }

    private Currency Currency { get; set; }

    ...
}
```

Когда понятие не имеет очевидной индивидуальности, это явное свидетельство того, что его допустимо реализовать в модели как объект-значение. Далее в этой главе будут представлены дополнительные примеры объектов-значений, которые позволят вам лучше понять эту мысль. В частности, вы узнаете, какие определяющие характеристики имеют большое значение для успешного применения объектов-значений.

Улучшение ясности

Основной целью философии DDD является ясное отражение важнейших бизнес-правил и предметной логики. Элементарные типы, такие как целые числа и строки, сами по себе не очень хорошо подходят для этого. Несмотря на возможность наглядно реализовать понятия с помощью элементарных типов, большинство практиков DDD настоятельно рекомендуют не поступать так. Элементарные типы должны заключаться в согласованные объекты-значения, четко представляющие понятия, которые они моделируют.

В листинге 15.2 показан объект, представляющий текущую выигравшую заявку на интернет-аукционе. Для представления суммы заявки используется целое число.

Листинг 15.2. Объект, опирающийся на неясный элементарный тип

```
public class WinningBid
{
    ...

    public int Price { get; private set; }

    ...
}
```

Представление суммы заявки в виде целого числа, как показано в листинге 15.2, нельзя признать оптимальным выбором по двум основным причинам. Во-первых, целое число ничего не говорит о том, что является ценой в данной предметной области, — оно никак не ограничивает диапазон допустимых значений и не отражает, в какой валюте или в каких единицах измерения выражена данная цена. Это обильный источник неоднозначности, скрывающий важные детали предметной области.

Когда сумма заявки моделируется в виде простого целого числа, увеличивается риск, что родственные предметные понятия, вместо того чтобы быть собранными в одном месте, будут разбросаны по всей предметной области из-за невозможности (или бессмысленности) придания поведения элементарным классам.

Реализация объекта-значения `Price` в листинге 15.3 показывает, что для предметной области интернет-аукциона правила увеличения суммы заявки играют важную роль. Как можно заметить, преимущество объекта-значения `Price` заключается в тесной группировке всех функций для работы с ценой: `BidIncrement()` и `CanBeExceededBy()`. Такой подход помогает выразить правила, касающиеся цены в этой предметной области, и гарантировать их соблюдение. Всякий раз, когда речь заходит о цене, разработчикам и специалистам в предметной области достаточно только взглянуть на один этот класс, чтобы понять, что имеется в виду под словом «цена» и какие правила действуют в отношении нее.

Листинг 15.3. Использование объекта-значения для увеличения ясности

```
public class Price
{
    public Price(Money amount)
    {
        if (amount == null)
            throw new ArgumentNullException("Amount cannot be null");

        Amount = amount;
    }

    public Money Amount { get; private set; }

    public Money BidIncrement()
    {
        if (Amount.IsGreaterThanOrEqualTo(new Money(0.01m))
            && Amount.IsLessThanOrEqualTo(new Money(0.99m)))
            return Amount.add(new Money(0.05m));

        if (Amount.IsGreaterThanOrEqualTo(new Money(1.00m))
            && Amount.IsLessThanOrEqualTo(new Money(4.99m)))
            return Amount.add(new Money(0.20m));

        if (Amount.IsGreaterThanOrEqualTo(new Money(5.00m))
            && Amount.IsLessThanOrEqualTo(new Money(14.99m)))
            return Amount.add(new Money(0.50m));
    }
}
```

```
        return Amount.add(new Money(1.00m));
    }

    public bool CanBeExceededBy(Money offer)
    {
        return offer.IsGreaterThanOrEqualTo(BidIncrement());
    }
}
```

Как показывает листинг 15.3, благодаря обертыванию элементарных типов предметными понятиями, система типов начинает отражать данные в предметную область, а все неясности исчезают. Как следствие, общение со специалистами в предметной области станет более ясным, а предметные понятия — более очевидными для любого, кто возьмется читать программный код. Кроме того, вся функциональность, связанная с единственным понятием, будет сосредоточена в одном месте. В данном случае правила увеличения цены и определения возможности превышения цены другой ценой однозначно сгруппированы с понятием цены.

ПРИМЕЧАНИЕ

В главе 16 «Сущности» вы узнаете, что добавление поведения в объекты-значения позволяет сохранить сущности более сосредоточенными. В сценарии с заявкой поведение, имеющее отношение к понятию цены, можно было бы выделить в класс `WinningBid` и наделить его дополнительными обязанностями.

Листинг 15.3 также иллюстрирует, что объекты-значения зачастую дробятся на более мелкие компоненты. Объект-значение `Price` сам использует связанное понятие для представления денег — объект-значение `Money`, используя экземпляр `Money` для выражения суммы заявки. Определение объекта-значения `Money` вы увидите далее в этой главе.

Характерные особенности

Практически полная самодостаточность — вот что делает объекты-значения такими простыми в использовании. Подобно приверженцам функционального программирования, практики DDD обожают объекты-значения, потому что они являются неизменяемыми, свободны от побочных эффектов и легко поддаются тестированию. В дальнейших примерах вы увидите, что они обладают некоторыми характерными, достаточно важными особенностями, о которых стоит узнать.

Отсутствие индивидуальности

Единственной важнейшей характеристикой объектов-значений, о которой следует помнить, является отсутствие у них индивидуальности. Объекты-значения играют важную роль, сообщая некоторую информацию о другом объекте. Какой рост указан в сущности `Person`? Какое количество лет безаварийной езды указано в сущности `InsurancePolicy`? Какой вес указан в сущности `Fruit`? Как видите, из

этих примеров можно понять, почему не имеет смысла говорить об индивидуальности объектов-значений.

Ввиду того что объекты-значения не имеют индивидуальности и служат для описания других понятий предметной области, обычно в первую очередь определяются сущности и лишь потом, когда станет понятно, какие типы значений необходимы, определяются соответствующие объекты-значения. В общем случае объекты-значения приобретают смысл только в контексте сущности.

Сравнение объектов-значений является важнейшей операцией, даже при том что они не имеют индивидуальности. Эта операция основывается на понятии равенства атрибутов, или значений.

ПРИМЕЧАНИЕ

Технически объекты-значения могут иметь уникальные идентификаторы (ID), полученные с использованием некоторых стратегий сохранения в базе данных. Это исключительно техническая деталь, связанная с их сохранением, которая не является признаком наличия индивидуальности в предметной области. Некоторые возможные способы сохранения объектов-значений будут показаны в конце этой главы.

Сравнение по атрибутам

Сущности считаются одинаковыми (равными), если имеют одинаковые идентификаторы (ID). Объекты-значения, напротив, считаются равными, если имеют одно и то же значение.

Если два объекта-значения *Currency* представляют одну и ту же сумму денег, они считаются равными вне зависимости от того, указывает каждая переменная на один и тот же объект или на разные объекты. Аналогично, если два объекта-значения *Temperature* представляют температуру 30 градусов Цельсия, они также считаются равными. В листинге 15.4 демонстрируется объект-значение *Meters*, используемый в предметной модели персональной системы для занятий фитнесом, которая следит, какое расстояние преодолел спортсмен. Далее, в листинге 15.5 показаны контрольные примеры, демонстрирующие, как определяется равенство на основе атрибутов.

Листинг 15.4. Объект-значение, определяющий понятие равенства на основе атрибутов

```
public class Meters
{
    public Meters(decimal distanceInMeters)
    {
        if (distanceInMeters < (decimal)0.0)
            throw new DistancesInMetersCannotBeNegative();

        this.DistanceInMeters = distanceInMeters;
    }

    protected decimal DistanceInMeters { get; private set; }
```

```

public Yards ToYards()
{
    return new Yards(DistanceInMeters * (decimal)1.0936133);
}

public Kilometers ToKilometers()
{
    return new Kilometers(DistanceInMeters / 1000);
}

public Meters Add(Meters meters)
{
    return new Meters(
        this.DistanceInMeters + meters.DistanceInMeters
    );
}

public bool IsLongerThan(Meters meters)
{
    return this.DistanceInMeters > meters.DistanceInMeters;
}

public override bool Equals(object obj)
{
    var m = obj as Meters;
    if (m == null) return false;

    return ToTwoDecimalPlaces(m.DistanceInMeters)
        == ToTwoDecimalPlaces(DistanceInMeters);
}

private decimal ToTwoDecimalPlaces(decimal distanceInMeters)
{
    return Math.Round(
        distanceInMeters, 2, MidpointRounding.AwayFromZero
    );
}
}

```

Листинг 15.5. Контрольные примеры, демонстрирующие равенство на основе атрибутов

```

[TestMethod]
public void Same_distances_are_equal_even_if_different_references()
{
    var oneMeter = new Meters((decimal)1);
    var oneMeterX = new Meters((decimal)1);
    Assert.AreEqual(oneMeter, oneMeterX);

    var fiftyPoint25 = new Meters((decimal)50.25);
    var fiftyPoint25X = new Meters((decimal)50.25);
    Assert.AreEqual(fiftyPoint25, fiftyPoint25X);
}

```

В листинге 15.4 видно, что объект-значение `Meters` переопределяет метод `Equals()` для реализации сравнения по атрибутам. По умолчанию в C# (и в других языках, таких как Java) два объекта считаются равными, если ссылки на них указывают на один и тот же объект в памяти. В случае с объектами-значениями нужно сравнивать не ссылки на них, а их предметные значения. В листинге 15.4 можно видеть, что реализация `Equals()` соответствует этому требованию, возвращая истинное значение, которое сообщает о равенстве двух объектов, если они представляют одно и то же расстояние в метрах с точностью до двух знаков после запятой (точность может меняться в зависимости от правил, действующих в предметной области, а также от контекста).

Листинг 15.4 содержит лишь минимальный объем кода, необходимый для демонстрации сравнения по атрибутам. Чтобы получить возможность использовать все инструменты сравнения, имеющиеся в языке C#, нужно переопределить еще несколько методов, используемых компилятором и средой выполнения в разных ситуациях. Реализация всех этих методов во всех объектах-значениях может оказаться весьма утомительным занятием, особенно если некоторые методы в действительности не будут использоваться. Как показано в листинге 15.6, есть возможность определить базовый класс, который избавит от ненужной рутины и повторяющегося кода.

Листинг 15.6. Базовый класс `ValueObject`, предоставляющий шаблонную реализацию сравнения

```
public abstract class ValueObject<T> where T : ValueObject<T>
{
    protected abstract IEnumerable<object> GetAttributesToIncludeInEqualityCheck();

    public override bool Equals(object other)
    {
        return Equals(other as T);
    }

    public bool Equals(T other)
    {
        if (other == null)
        {
            return false;
        }
        return GetAttributesToIncludeInEqualityCheck()
            .SequenceEqual(other.GetAttributesToIncludeInEqualityCheck());
    }

    public static bool operator ==(ValueObject<T> left, ValueObject<T> right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(ValueObject<T> left, ValueObject<T> right)
    {

```

```
        return !(left == right);
    }

    public override int GetHashCode()
    {
        int hash = 17;
        foreach (var obj in this.GetAttributesToIncludeInEqualityCheck())
            hash = hash * 31 + (obj == null ? 0 : obj.GetHashCode());

        return hash;
    }
}
```

Для обеспечения поддержки вашими классами всех операций сравнения, имеющих в языке C#, нужно реализовать `Equals()`, `GetHashCode()` и определить перегруженные версии операторов `==` и `!=`. Использование базового класса, подобного тому, что представлен в листинге 15.6, упростит задачу поддержки сравнения. Вам останется только реализовать `GetAttributesToIncludeInEqualityCheck()` в каждом подклассе, как показано в листинге 15.7. Если вы готовы к дополнительным сложностям, попробуйте реализовать альтернативное решение на основе механизма рефлексии¹ (reflection).

Листинг 15.7. Альтернативная реализация объекта-значения с использованием базового класса

```
public class Meters : ValueObject<Meters>
{
    ...

    protected override IEnumerable<object>
    GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object> { DistanceInMeters };
    }
}
```

В листинге 15.7 показана альтернативная реализация `Meters` с той же семантикой и поведением, которая дополнительно поддерживает операторы `==` и `!=`. Это усовершенствование избавляет от необходимости повторять и повторять шаблонный код, который теперь помещен в базовый класс `ValueObject`.

ПРИМЕЧАНИЕ

Некоторые языки программирования имеют встроенную поддержку классов, подобных объектам-значениям. В Scala, например, `case`-классы имеют встроенную поддержку сравнения по атрибутам. По этой причине к выбору языка программирования определенно стоит подойти основательно.

¹ Его часто также называют механизмом отражения, или интроспекции. — *Примеч. пер.*

Разнообразие возможностей

Объекты-значения должны экспортировать максимально выразительные предметно-ориентированные возможности и скрывать состояние. Это было продемонстрировано в листинге 15.4, где объект-значение `Meters` экспортирует функции: `ToFeet()`, `ToKilometers()`, `Add()` и `IsLongerThan()`. Также обратите внимание, как он скрывает свое состояние `DistanceInMeters`. Как правило, все элементарные значения должны объявляться как закрытые (`private`) или защищенные (`protected`). И только при наличии веских причин можно нарушить принцип инкапсуляции и экспортировать состояние как общедоступное (`public`). Но прежде попробуйте добавить в объект-значение метод, реализующий необходимую функциональность.

Как не раз повторялось в этой книге, функциональным возможностям предметных моделей должно уделяться особое внимание как одному из важнейших аспектов создания и развития предметных моделей, четко представляющих предметные понятия, которые они моделируют. Это в равной степени справедливо для объектов-значений и других типов предметных объектов, которые будут представлены в книге далее.

Согласованность

Как описательное понятие, обычно определяющее некоторую количественную характеристику, объекты-значения часто согласованно включают в себя не только само значение, но и единицу измерения. Вы могли увидеть это в примере с объектом-значением `Money`, включающим сумму и валюту.

Однако согласованность не всегда означает включение значения и единицы измерения. Число согласованных полей может быть любым. Например, объект-значение `Color` может иметь свойства `Red`, `Green` и `Blue`.

Неизменяемость

После создания объекты-значения никогда не изменяются. Вместо этого все попытки изменить их значения должны приводить к созданию новых экземпляров с требуемыми значениями. Это объясняется тем, что неизменяемые объекты проще поддаются анализу и имеют меньше нежелательных побочных эффектов.

Классическим примером неизменяемости и вообще хорошим образцом для подражания при создании объектов-значений является класс `DateTime` в .NET. Модульные тесты в листинге 15.8 наглядно показывают, как любые операции, которые, казалось бы, изменяют состояние объекта, включая `AddMonths()` и `AddYears()`, в действительности возвращают совершенно новый объект `DateTime`.

Листинг 15.8. `DateTime` — отличный пример неизменяемости

```
[TestClass]
public class DateTime_immutability_specs
{
    [TestMethod]
```

```

public void AddMonths_creates_new_immutable_DateTime()
{
    var jan1st = new DateTime(2014, 01, 01);
    var feb1st = jan1st.AddMonths(1);

    // первый объект не изменился
    Assert.AreEqual(new DateTime(2014, 01, 01), jan1st);

    // второй объект – новый неизменяемый экземпляр
    Assert.AreEqual(new DateTime(2014, 02, 01), feb1st);
}

[TestMethod]
public void AddYears_creates_new_immutable_DateTime()
{
    var jan2014 = new DateTime(2014, 01, 01);
    var jan2015 = jan2014.AddYears(1);
    var jan2016 = jan2015.AddYears(1);

    // первый объект не изменился
    Assert.AreEqual(new DateTime(2014, 01, 01), jan2014);

    // второй объект не изменился
    Assert.AreEqual(new DateTime(2015, 01, 01), jan2015);
    Assert.AreEqual(new DateTime(2016, 01, 01), jan2016);
}
}

```

Поддержка неизменяемости реализуется достаточно просто. Сначала нужно решить, каким образом будет экспортироваться состояние. Одним из вариантов может быть определение переменных экземпляра, доступных только для чтения, как показано в листинге 15.9. Другой подход — объявление свойств, которые устанавливаются конструктором и никогда не изменяются, как это демонстрирует предыдущий листинг. Оба подхода часто используются на практике, и выбор между ними обычно зависит от личных предпочтений или от соглашений, принятых в проекте. Однако в исключительных случаях выбор одного из вариантов может быть продиктован используемыми фреймворками.

Помимо самого состояния, вы должны экспортировать методы с описательными именами, выражающими создание нового значения в предметно-ориентированных терминах. Для `DateTime` в число таких методов входят уже обсуждавшиеся `AddMonths()` и `AddYears()`; для объекта-значения `Money` из листинга 15.9 такими методами являются `Add()` и `Subtract()`.

Листинг 15.9. Превращение объектов-значений в неизменяемые объекты

```

public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
    {
    }
}

```

```
public Money(decimal value)
{
    Value = value;
}

public Money Add(Money money)
{
    return new Money(Value + money.Value);
}

public Money Subtract(Money money)
{
    return new Money(Value - money.Value);
}

protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() { Value };
}
}
```

Объект-значение `Money` из листинга 15.9 имеет два метода, имена которых выглядят так, будто они изменяют состояние объекта: `Add()` и `Subtract()`. Если же взглянуть на реализацию, можно убедиться, что это не так, потому что оба метода возвращают новый экземпляр `Money` с измененным значением, при этом исходный экземпляр не изменяется. Поддержка неизменяемости обеспечивает также комбинированность, как показано в следующем примере.

Комбинируемость

Значения часто бывают представлены числами, поэтому в большинстве случаев их можно комбинировать, чтобы создать новое значение. Как вы могли видеть в предыдущем примере, объекты `Money` можно складывать, чтобы создать новый объект `Money`. Такая комбинируемость является характерной особенностью объектов-значений в целом. Поэтому когда в беседах со специалистами в предметной области вы услышите о возможности комбинирования двух экземпляров определенного понятия, это можно рассматривать как явный признак того, что это понятие, возможно, потребуется смоделировать в виде объекта-значения.

ПРИМЕЧАНИЕ

Под комбинируемостью в данном контексте подразумевается возможность комбинирования двух объектов-значений. Не забывайте при этом, что объекты-значения являются неизменяемыми и результатом будет новый объект-значение с желаемым значением. Исходные объекты не изменятся.

Часто объекты-значения, представляющие числовую величину или количество, можно комбинировать с помощью таких операций, как сложение, вычитание и умножение, как показано в предыдущем примере, демонстрирующем поддержку

неизменяемости. Для большей выразительности можно переопределить эти операции на уровне объектов, что допускается во многих языках программирования. В листинге 15.10 показано, как изменить объект-значение `Money`, чтобы добавить в него поддержку операторов `+` (плюс) и `-` (минус).

Листинг 15.10. Поддержка комбинированности в объектах-значениях

```
public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
    {
    }

    public Money(decimal value)
    {
        Value = value;
    }

    public Money Add(Money money)
    {
        return new Money(Value + money.Value);
    }

    public Money Subtract(Money money)
    {
        return new Money(Value - money.Value);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object>() { Value };
    }

    public static Money operator + (Money left, Money right)
    {
        return new Money(left.Value + right.Value);
    }

    public static Money operator - (Money left, Money right)
    {
        return new Money(left.Value - right.Value);
    }
}
```

Переопределение операторов `+` и `-` легко реализуется в C#, как это видно в листинге 15.10. В листинге 15.11 демонстрируется использование этих операторов для комбинирования неизменяемых объектов `Money` с получением новых экземпляров.

Листинг 15.11. Комбинирование неизменяемых объектов-значений с получением новых экземпляров

```
[TestClass]
public class Combining_money_tests
{
    [TestMethod]
    public void Money_supports_native_addition_syntax()
    {
        var m = new Money(200);
        var m2 = new Money(300);

        var combined = m + m2;

        Assert.AreEqual(new Money(500), combined);
    }

    [TestMethod]
    public void Money_supports_native_subtraction_syntax()
    {
        var m = new Money(50);
        var m2 = new Money(49);

        var combined = m - m2;

        Assert.AreEqual(new Money(1), combined);
    }
}
```

Автоматическая проверка

Объекты-значения никогда не должны находиться в недопустимом состоянии. Искключительная ответственность за соблюдение этого требования лежит на них самих. На практике это означает, что когда создается экземпляр объекта-значения, конструктор должен возбудить исключение, если его аргументы не соответствуют предметным правилам. Например, в отношении модели денег в виде объекта-значения `Money` в приложении электронной коммерции могут действовать два важных правила:

- Все денежные суммы должны иметь точность до двух знаков после запятой.
- Все денежные суммы должны выражаться положительными значениями.

Эти два правила применяются ко всем экземплярам `Money` в данной предметной области и никогда не должны нарушаться. В листинге 15.12 показан объект-значение `Money`, который обеспечивает соблюдение этих требований в конструкторе, гарантируя невозможность создания нового экземпляра в недопустимом состоянии.

Листинг 15.12. Объект-значение с автоматической проверкой

```
public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
```

```

{
}

public Money(decimal value)
{
    Validate(value);
    Value = value;
}

private void Validate(decimal value)
{
    if (value % 0.01m != 0)
        throw new MoreThanTwoDecimalPlacesInMoneyValueException();

    if (value < 0)
        throw new MoneyCannotBeANegativeValueException();
}

public Money Add(Money money)
{
    return new Money(Value + money.Value);
}

protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() { Value };
}
}

```

Первым делом конструктор вызывает метод `Validate()`, который осуществляет автоматическую проверку. Разумная реализация не требует больших усилий. Как демонстрирует листинг 15.12, во многих случаях достаточно просто убедиться, что состояние объекта-значения находится в допустимых границах. Если это не так, можно просто возбудить исключение и сразу же прервать создание нового экземпляра.

Однако существуют другие подходы к осуществлению проверки, которые могут лучше соответствовать вашим предпочтениям. Выразительность и гибкость подходов может отличаться исходя из контекста, поэтому вам стоит познакомиться с ними поближе. Первый альтернативный подход заключается в организации проверки внутри фабричного метода, как показано в листинге 15.13.

Листинг 15.13. Проверка значения внутри фабричного метода

```

public class Money : ValueObject<Money>
{
    // ..

    public static Money Create(decimal amount)
    {
        if (amount % 0.01m != 0)
            throw new MoreThanTwoDecimalPlacesInMoneyValueException();
    }
}

```

```
        if (amount < 0)
            throw new MoneyCannotBeANegativeValueException();

        return new Money(amount);
    }
}
```

Обратите внимание, что метод `Create()` в листинге 15.13 объявлен статическим (`static`). Это фабричный метод, представляющий альтернативу проверке в конструкторе `Money`. Такой подход часто используется, когда объекты-значения могут быть созданы с разными состояниями исходя из контекста. Например, в некоторых сценариях может пригодиться возможность создавать отрицательные суммы.

Применение фабричных методов означает, что проверку можно обойти и создать экземпляр с помощью ключевого слова `new`. Поэтому используйте этот подход с большой осторожностью. Если возможны ситуации, когда в разных контекстах применяются разные правила проверки, вам определенно стоит прочитать введение в микротипы, о которых пойдет речь далее в этой главе.

Другой подход к организации проверки в объектах-значениях — использование программных контрактов (`code contracts`), которые обеспечивают лучшую гибкость и выразительность за счет дополнительной сложности. Контракты можно использовать как дополнение к любому из подходов, представленных выше. Листинг 15.14 иллюстрирует использование программных контрактов в сочетании с проверкой в конструкторе.

Листинг 15.14. Автоматическая проверка в конструкторе с применением программных контрактов

```
public class Name : ValueObject<Name>
{
    public readonly string firstName;
    public readonly string surname;

    public Name(string firstName, string surname)
    {
        Check.that(firstName.is_not_empty()).on_constraint_failure(() =>
        {
            throw new ApplicationException("You must specify a first name.");
        });

        Check.that(surname.is_not_empty()).on_constraint_failure(() =>
        {
            throw new ApplicationException("You must specify a surname.");
        });

        this.firstName = firstName;
        this.surname = surname;
    }

    // ..
}
```

При выборе программных контрактов от лишних сложностей можно избавиться, определив контракты и вспомогательные объекты для многократного использования. В листинге 15.15 показано несколько простых программных контрактов, которые применяются в листинге 15.14. Вы можете создать свою библиотеку со вспомогательными объектами или воспользоваться существующими библиотеками программных контрактов, например предлагаемыми компанией Microsoft ([http://msdn.microsoft.com/ru-ru/library/dd264808\(v=vs.110\).aspx](http://msdn.microsoft.com/ru-ru/library/dd264808(v=vs.110).aspx)).

Листинг 15.15. Программные контракты для многократного использования

```
public static class StringExtensions
{
    public static bool is_not_empty(this String string_to_check)
    {
        return !String.IsNullOrEmpty(string_to_check);
    }
}

public class CheckConstraint
{
    private readonly bool _assertion;

    public CheckConstraint(bool assertion)
    {
        _assertion = assertion;
    }

    public void on_constraint_failure(Action onFailure)
    {
        if (!_assertion) onFailure();
    }
}

public sealed class Check
{
    public static CheckConstraint that(bool assertion)
    {
        return new CheckConstraint(assertion);
    }
}
```

Простота тестирования

Неизменяемость, согласованность и способность к комбинированию — вот три основных качества объектов-значений, которые делают их такими простыми для тестирования на выразительном предметно-ориентированном языке. Неизменяемость избавляет от необходимости использовать объекты-имитации (mocks) или проверять действие побочных эффектов, согласованность позволяет провести всестороннее тестирование того или иного понятия в полной изоляции, а способность к комбинированию дает возможность выражать отношения между разными значениями.

В этом разделе вы уже видели несколько примеров модульных тестов. Однако они демонстрировались в контексте других характеристик, поэтому кто-то, возможно, не успел оценить их простоту. В листинге 15.16 приводится еще один короткий пример, демонстрирующий простоту тестирования объектов-значений и то, как легко можно проверить ошибочные условия без применения объектов-имитаций.

Листинг 15.16. Простота тестирования объектов-значений

```
[TestClass]
public class Name_validation_tests
{
    [TestMethod]
    public void First_names_cannot_be_empty()
    {
        try
        {
            var name = new Name("", "Torvalds");
        }
        catch (ApplicationException e)
        {
            Assert.AreEqual("You must specify a first name.", e.Message);
            return;
        }
        Assert.Fail("No ApplicationException was thrown");
    }

    [TestMethod]
    public void Surnames_cannot_be_empty()
    {
        try
        {
            var name = new Name("Linus", "");
        }
        catch (ApplicationException e)
        {
            Assert.AreEqual("You must specify a surname.", e.Message);
            return;
        }
        Assert.Fail("No ApplicationException was thrown");
    }
}
```

Первое, что бросается в глаза при рассмотрении тестов в листинге 15.16, — это их простота в сравнении с тестами, реализующими проверку других объектов, таких как прикладные службы. Это в значительной степени объясняется отсутствием побочных эффектов и изменяемого состояния. Вам часто придется слышать восторженные отзывы об этих характеристиках от сторонников функционального программирования. По сути, объекты-значения сами являются функциональной концепцией (или, по крайней мере, они очень близки к этому).

ПРИМЕЧАНИЕ

В контексте функционального программирования для описания методов или функций без побочных эффектов используется интересный термин «ссылочная прозрачность» (referential transparency). Многим программистам стоит получить узнать, что такое ссылочная прозрачность и функциональное программирование. Языки, такие как F#, Scala и Haskell, продолжают приобретать все большую популярность. Отличным введением в Haskell может служить статья, описывающая суть и принципы функционального программирования (http://www.haskell.org/haskellwiki/Functional_programming).

Общие шаблоны моделирования

За годы существования методологии практиками DDD была создана небольшая коллекция шаблонов для работы с объектами-значениями. Главным образом они нацелены на повышение выразительности и ясности, но некоторые позволяют получить другие небольшие преимущества, включая удобство в сопровождении. В этом разделе рассматриваются три шаблона. Все они настолько просты, что вы сразу после знакомства с ними сможете приступить к их применению и задумать-ся о проектировании собственных.

Статические фабричные методы

Использование статических фабричных методов (static factory methods) — популярный прием сокрытия сложностей, связанных с созданием объекта, за простым и выразительным интерфейсом. Отличным примером применения этого шаблона может служить класс `TimeSpan` в .NET с его статическими фабричными методами `FromDays()`, `FromHours()` и `FromMilliseconds()`. Эти альтернативы более выразительны и однозначны в сравнении с конструктором, принимающим пять целочисленных характеров, как иллюстрирует листинг 15.17.

Листинг 15.17. Статические фабричные методы могут быть более выразительными и скрывать сложности

```
[TestMethod]
public void TimeSpan_factory_methods()
{
    var sixDays = TimeSpan.FromDays(6);
    var threeHours = TimeSpan.FromHours(3);
    var twoMillis = TimeSpan.FromMilliseconds(2);

    var sixDaysx = new TimeSpan(6, 0, 0, 0, 0);
    var threeHoursx = new TimeSpan(0, 3, 0, 0, 0);
    var twoMillisx = new TimeSpan(0, 0, 0, 0, 2);

    Assert.AreEqual(sixDays, sixDaysx);
    Assert.AreEqual(threeHours, threeHoursx);
    Assert.AreEqual(twoMillis, twoMillisx);
}
```

Статические фабричные методы — это стилистический шаблон, который вы можете использовать или не использовать, как вам заблагорассудится. В листинге 15.18 показано, какие изменения можно было бы внести в объект-значение `Height`, чтобы добавить в него по одному статическому фабричному методу для каждой единицы измерения. Эти методы делают объект-значение более выразительным, простым в использовании и удобным в сопровождении, потому что клиентам больше не придется связывать себя необходимостью использовать перечисление `MeasurementUnit`.

Листинг 15.18. Статические фабричные методы могут улучшать выразительность и удобство сопровождения

```
public class Height
{
    public Height(int size, MeasurementUnit unit)
    {
        this.Size = size;
        this.Unit = unit;
    }

    public int Size { get; private set; }

    public MeasurementUnit Unit { get; private set; }

    // ..

    public static Height FromFeet(int feet)
    {
        return new Height(feet, MeasurementUnit.Feet);
    }

    public static Height FromMetres(int metres)
    {
        return new Height(metres, MeasurementUnit.Metres);
    }
}
```

Микротипы

Уход от использования элементарных типов помогает более четко обозначить цель и уменьшить неоднозначность программного кода. Об этом уже говорилось выше в данной главе. Однако этот принцип можно развить еще дальше, применив шаблон с названием «микротипы» (*micro types*), суть которого заключается в обертывании и без того выразительных типов еще более выразительными типами. Следует уточнить, что обертываемые типы не могут быть элементарными типами; это должны быть уже явно выраженные понятия, обертывающие элементарные типы. Данный шаблон может пригодиться, например, чтобы увеличить ясность в определенных контекстах и избежать ошибок.

Рассмотрим порядок применения шаблона «микротипы» сначала на примере предметной службы `OvertimeCalculator`, представленной в листинге 15.19. Обратите внимание на два параметра с типами `HoursWorked` и `ContractedHours`.

Листинг 15.19. Предметная служба, принимающая только микротипы

```
public class OvertimeCalculator
{
    public OvertimeHours Calculate(HoursWorked worked, ContractedHours contracted)
    {
        var overtimeHours = worked.Hours - contracted.Hours;
        return new OvertimeHours(overtimeHours);
    }
}
```

Оба типа, `HoursWorked` и `ContractedHours`, используемые в листинге 15.19, — это микротипы, обертывающие объект-значение `Hours`. `OvertimeHours` — это возвращаемый микротип, который также является лишь контекстной оберткой вокруг `Hours`. Определения всех четырех классов приводятся в листинге 15.20.

Листинг 15.20. Микротипы, обертывающие и без того выразительный объект-значение

```
// основной объект-значение
public class Hours : ValueObject<Hours>
{
    public readonly int Amount;

    public Hours(int amount)
    {
        this.Amount = amount;
    }

    public static Hours operator - (Hours left, Hours right)
    {
        return new Hours(left.Amount - right.Amount);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Amount };
    }
}

// микротипы
public class HoursWorked : ValueObject<HoursWorked>
{
    public readonly Hours Hours;

    public HoursWorked(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}
```



```

    }
}

public class ContractedHours : ValueObject<ContractedHours>
{
    public readonly Hours Hours;

    public ContractedHours(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}

public class OvertimeHours : ValueObject<OvertimeHours>
{
    public readonly Hours Hours;

    public OvertimeHours(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}

```

Как можно заметить в листингах 15.20 и 15.21, типы `HoursWorked` и `ContractedHours` не добавляют никаких дополнительных возможностей или данных. Мы легко могли бы определить метод `OvertimeCalculator.Calculate()` так, чтобы он принимал два экземпляра `Hours`. Но такая реализация полагается на явные имена параметров и ожидает получить от вызывающего кода два объекта `Hours` в правильном порядке. Используя микротипы, можно заставить работать систему типов так, чтобы одновременно увеличить ясность программного кода и исключить человеческие ошибки.

Листинг 15.21. Модульный тест, демонстрирующий контекстное применение микротипов

```

[TestClass]
public class Micro_types_example_tests
{
    [TestMethod]
    public void Calculates_overtime_hours_as_hours_additional_to_contracted()
    {
        var hoursWorked = new Hours(40);
    }
}

```

```

var contractedHours = new Hours(35);

// обернуть микротипами для большей ясности
var hoursWorkedx = new HoursWorked(hoursWorked);
var contractedHoursx = new ContractedHours(contractedHours);

var fiveHours = new Hours(5);
var fiveHoursOvertime = new OvertimeHours(fiveHours);

var calc = new OvertimeCalculator();
var result = calc.Calculate(hoursWorkedx, contractedHoursx);

Assert.AreEqual(fiveHoursOvertime, result);
}
}

```

Использование микротипов — далеко не самый лучший прием в промышленном программировании. В действительности это весьма спорное решение. Одни утверждают, что микротипы помогают писать более ясный, более пригодный для компоновки программный код, тогда как другие считают, что микротипы добавляют слишком много ненужных уровней косвенности, вызывающих раздражение. Вам решать, использовать шаблон «микротипы» или нет.

Отказ от коллекций

Некоторые практики DDD считают, что всегда следует избегать использования коллекций объектов-значений. Это объясняется тем, что простые коллекции недостаточно точно отражают предметные понятия. Также считается, что при извлечении элементов из коллекции объектов-значений необходимо использовать какие-либо формы идентификации этих элементов, что явно нарушает принцип отсутствия индивидуальности у объектов-значений. И снова данная практика не является универсальной для всего сообщества, тем не менее не следует избегать ее без веских на то причин.

Примером, демонстрирующим нехватку ясности при использовании коллекции объектов-значений, может служить сущность *Customer*, представленная в листинге 15.22, имеющая множество объектов-значений *PhoneNumber*.

Листинг 15.22. Сущность с коллекцией объектов-значений

```

public class Customer
{
    public Customer(Guid id)
    {
        this.Id = id;
    }

    public Guid Id { get; private set; }

    public IEnumerable<PhoneNumber> PhoneNumbers { get; set; }
}

```

```
// ..
}

public class PhoneNumber : ValueObject<PhoneNumber>
{
    public readonly string Number;

    public PhoneNumber(string number)
    {
        this.Number = number;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Number };
    }

    // ..
}
```

Здесь имеется множество объектов `PhoneNumber`, составляющих коллекцию `PhoneNumbers`. Для чего они используются? Может быть, они представляют номера домашнего и мобильного телефонов? Возможно, в коллекции имеется также номер для срочной связи или номер рабочего телефона. Использование коллекции не дает никакой ясности в этом вопросе. Но в данной предметной области коллекция объектов-значений `PhoneNumber` структурирована в соответствии с требованием предприятия, согласно которому каждый клиент должен указать номера домашнего, мобильного и рабочего телефонов.

В листинге 15.23 представлен более удачный подход к моделированию важного предметного понятия, определяющего номера домашнего, мобильного и рабочего телефонов, с использованием более очевидной структуры. Кроме того, такая организация упрощает изменение любого из имеющихся номеров без применения поиска по идентификатору или приема прямой индексации массива.

Листинг 15.23. Сущность с выразительным объектом-значением, замещающим коллекцию

```
public class Customer
{
    public Customer(Guid id)
    {
        this.Id = id;
    }

    public Guid Id { get; private set; }

    public PhoneBook PhoneNumbers { get; set; }

    // ..
}
```

```

}

public class PhoneBook : ValueObject<PhoneBook>
{
    public readonly PhoneNumber HomeNumber;
    public readonly PhoneNumber MobileNumber;
    public readonly PhoneNumber WorkNumber;

    public PhoneBook(PhoneNumber homeNum, PhoneNumber mobileNum,
                    PhoneNumber workNum)
    {
        this.HomeNumber = homeNum;
        this.MobileNumber = mobileNum;
        this.WorkNumber = workNum;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { HomeNumber, MobileNumber, WorkNumber };
    }
}

public class PhoneNumber : ValueObject<PhoneNumber>
{
    public readonly string Number;

    public PhoneNumber(string number)
    {
        this.Number = number;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Number };
    }

    // ..
}

```

Измененная сущность **Customer** из листинга 15.23 теперь хранит объекты-значения **PhoneNumber** в другом объекте-значении — **PhoneBook**. Номер каждого телефона — домашнего, рабочего и мобильного — теперь доступен напрямую, и нет необходимости использовать поиск по идентификатору. Это важно, потому что, как уже не раз упоминалось, объекты-значения не имеют индивидуальности.

Но самое важное отличие реализации в листинге 15.23 от реализации в листинге 15.22 заключается в более четком отражении предметных понятий. Сущности **Customer** имеют номера домашнего, рабочего и мобильного телефонов. Эта важная предметная структура заложена в типы, и ее соблюдение обеспечивается системой типов.

Сохранение

Самым сложным аспектом работы с объектами-значениями является, пожалуй, их сохранение. Использование документоориентированных хранилищ данных, таких как RavenDB и EventStore, вызывает меньше всего проблем; эти технологии позволяют сохранять объекты-значения вместе с сущностями в одном документе. Однако использование баз данных SQL, с их традициями нормализации, допускает большее разнообразие возможных реализаций. Поэтому оставшаяся часть главы в основном будет посвящена использованию баз данных SQL.

ПРИМЕЧАНИЕ

Вы сможете опробовать, изменять и исследовать все примеры в этом разделе, если загрузите архив с примерами для данной главы.

NoSQL

Многие базы данных NoSQL хранят данные в денормализованном виде, что резко отличает их от баз данных SQL, где действуют строгие соглашения по нормализации. Применение баз данных NoSQL может оказаться выгодным для DDD, потому что позволяет хранить сущности — а иногда и целые агрегаты — в единственном документе. Проблемы, связанные с соединением таблиц, нормализацией данных и организацией отложенной загрузки в инструментах ORM, просто отсутствуют в документоориентированных моделях. В контексте объектов-значений это означает, что они могут храниться вместе со своими сущностями. Например, сущность `Customer` и объект-значение `Name`, представленный в листинге 15.24, можно сохранить в виде единого денормализованного документа в формате представления объектов JavaScript (JavaScript Object Notation, JSON), как показано в листинге 15.25.

Листинг 15.24. Сущность с объектами-значениями, которую требуется сохранять

```
public class Customer
{
    // ..

    public Guid Id { get; protected set; }

    public Name Name { get; protected set; }

    // ..
}

public class Name : ValueObject<Name>
{
    // ..

    public string FirstName { get; protected set; }
```

```
public string Surname { get; protected set; }

// ..
}
```

Листинг 15.25. Сущность с объектами-значениями в денормализованном документоориентированном формате JSON

```
{
  "Customer": {
    "Id": ..,
    "Name": {
      "FirstName": ..,
      "Surname": ..
    },
    // ..
  }
}
```

Многие документоориентированные базы данных NoSQL позволяют сохранять документы в формате JSON. В листинге 15.25 показано, как можно сохранить сущность `Customer` и ее объект-значение `Name` в такой базе данных в виде единого документа JSON. Документоориентированные базы данных, такие как RavenDB, обычно применяют это соглашение по умолчанию, а это означает, что от вас требуется лишь создать модель объекта, структурированную соответствующим образом.

Несмотря на то что внедрение объектов-значений в документы является распространенным соглашением, такой подход — лишь один вариант из множества. При необходимости объекты-значения можно сохранять в отдельных документах из соображений производительности.

SQL

Существует несколько вариантов сохранения объектов-значений в базе данных SQL. Можно последовать стандартным соглашениям SQL и нормализовать объекты-значения, определив для них отдельные таблицы, или же денормализовать их по аналогии с документоориентированным подходом, как показано в листинге 15.25. В этом разделе будут показаны примеры каждого из этих двух распространенных подходов. Следует отметить, что данные примеры являются лишь ориентирами. Существует множество разновидностей каждого из этих подходов, поэтому вы можете свободно экспериментировать и создавать свои варианты, наиболее соответствующие вашим потребностям.

Простая денормализация

На практике часто используется прием непосредственного сохранения содержимого объектов-значений в нестандартном формате. Это отличный выбор, когда нет желания создавать множество таблиц и писать длинные запросы, соединяющие эти таблицы.

Отличным кандидатом для непосредственного сохранения содержимого является объект-значение `DateTime`, так как его можно записать в базу данных SQL в простом текстовом виде. Не нужно создавать отдельную таблицу `DateTime` для сохранения каждого значения, используемого в приложении. Фактически `DateTime` является настолько распространенным значением, что для него в базах данных предусмотрен собственный тип. Аналогичным образом можно смоделировать другие объекты-значения, даже не имеющие соответствующих им типов в базе данных. Правда, для этого придется определить свой формат хранения и, возможно, научить свой фреймворк и инструмент ORM использовать его.

Определение формата хранения

Чтобы иметь возможность восстановить объект-значение из хранилища, требуется определить формат, уникально идентифицирующий каждое возможное значение. Затем объект-значение можно сохранить в этом формате в базе данных, а также извлечь из базы данных, преобразовав данные в этот формат.

Для преобразования объектов-значений в формат хранения часто используется прием переопределения метода `ToString()`. В связи с тем что формат уникально описывает объект-значение, он может быть крайне полезен при отладке. В листинге 15.26 показана измененная версия объекта `Name` с методом `ToString()`, возвращающим уникальное описание значения, которое он представляет, для поддержки возможности сохранения объекта в этом формате. Далее, в листинге 15.27, приводится несколько модульных тестов, демонстрирующих его работу.

Листинг 15.26. Переопределение `ToString()` для получения готового к сохранению уникального описания

```
public class Name : ValueObject<Name>
{
    public Name(string firstName, string surname)
    {
        this.FirstName = firstName;
        this.Surname = surname;
    }

    public string FirstName { get; protected set; }

    public string Surname { get; protected set; }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { FirstName, Surname };
    }

    public override string ToString()
    {
        return String.Format(
            "firstName:{0};surname:{1}", FirstName, Surname
        );
    }
}
```

Листинг 15.27. Пример представления объекта-значения

```
[TestMethod]
public void Each_value_has_a_unique_representation()
{
    var sallySmith = new Name("Sally", "Smith");
    Assert.AreEqual("firstName:Sally;;;surname:Smith", sallySmith.ToString());

    var billyJean = new Name("Billy", "Jean");
    Assert.AreEqual("firstName:Billy;;;surname:Jean", billyJean.ToString());
}
```

Определение формата хранения главным образом заключается в определении уникального представления для каждого возможного значения. Это самое меньшее, что необходимо, чтобы получить возможность сохранять объекты. Метод `ToString()` в листинге 15.26 возвращает удобочитаемое представление. В своих приложениях вы также можете предпочесть использовать удобочитаемые форматы, но иногда бывает желательно иметь более компактные форматы или даже форматы, обладающие другими качествами. Выбирая формат хранения, также имейте в виду, что в какой-то момент может понадобиться добавить фильтрацию значений в запросы SQL.

Еще одна особенность реализации из листинга 15.26 — переопределение метода `ToString()`. Этот прием дает двойную выгоду: данный метод можно использовать не только для сохранения объектов, но и для отладки. Но если потребуется разделить `ToString()` и формат хранения, можно определить перегруженную версию `ToString()` или создать другой метод, например `ToPersistenceFormat()`. Наиболее важная деталь заключается в том, чтобы метод, отвечающий за преобразование объекта в формат хранения, вызывался в момент выполнения операции сохранения либо вашим запросом SQL, либо используемым у вас инструментом ORM.

Сохранение значений

Данный пример демонстрирует, как сохранить объект-значение с помощью ORM (наиболее сложный случай). Если вы пишете запросы SQL сами, вам не составит труда передать представление своего объекта-значения в запрос `INSERT` или `UPDATE`.

Чтобы сохранить объект-значение, важно правильно подготовить инструмент ORM. При использовании NHibernate, например, можно создать свой класс, реализующий интерфейс `IUserType`, как показано в листинге 15.28. Большинство фреймворков ORM предоставляют похожие возможности.

Листинг 15.28. Сохранение объекта-значения вручную с использованием точки входа во фреймворк

```
public class NameValueObjectPersister : IUserType
{
    // ..

    public void NullSafeSet(IDbCommand cmd, object value, int index)
    {
        // параметр value – это экземпляр Name для сохранения
        var parameter = (IDataParameter)cmd.Parameters[index];
```



```
if (value == null)
{
    parameter.Value = DBNull.Value;
}
else
{
    // Name.ToString() вернет представление в формате для сохранения
    parameter.Value = value.ToString();
}
// ....
}
```

Интерфейс `IUserType`, объявленный во фреймворке `NHibernate`, позволяет разработчикам вручную управлять сохранением объектов определенных типов. В листинге 15.28 представлен класс `NameValueObjectPersister`, реализующий интерфейс `IUserType`, который управляет сохранением объектов-значений `Name`. Как можно заметить, логика, генерирующая значение для сохранения в базе данных, должна находиться в методе `IUserType.NullSafeSet()`.

ПРИМЕЧАНИЕ

Фреймворк `NHibernate` и некоторые другие инструменты объектно-реляционного отображения имеют встроенную поддержку объектов-значений, избавляющую от необходимости определять свои форматы хранения. Данная поддержка в `NHibernate` реализована в виде понятия Компонент (`Component`), как демонстрируется в блоге `NHibernate` (<http://nhibernate.info/blog/2008/09/17/value-objects.html>). Примеры в этом разделе не используют данную возможность и демонстрируют более универсальное решение, пригодное для любой технологии. Однако вам определенно стоит исследовать встроенную поддержку объектов-значений в вашем инструменте ORM, потому что она может помочь сэкономить уйму времени.

ПРИМЕЧАНИЕ

Чтобы иметь возможность опробовать примеры на основе `NHibernate`, приведенные в оставшейся части главы, нужно включить `NHibernate` в проект как зависимость. Для этого достаточно добавить `NuGet`-пакеты `NHibernate` и `FluentNHibernate`.

Лучшим руководством по `NHibernate` для начинающих является официальная документация (<http://nhibernate.info/doc/tutorials/first-nh-app/your-first-nhibernate-based-application.html>). Точно так же лучшим источником информации о `FluentNHibernate` является официальная документация для `FluentNHibernate` (<https://github.com/jagregory/fluent-nhibernate/wiki/Getting-started>).

Парсинг значений при загрузке

После сохранения объекта-значения с помощью инструмента ORM или собственного запроса остается нерешенной задача загрузки представления объекта из хранилища и его преобразования обратно в объект-значение. Существует несколько подходов к решению этой задачи. Один из них состоит в том, чтобы определить

конструктор, принимающий строковое представление объекта-значения. Другой вариант — создать промежуточный компонент, который будет выполнять парсинг строки и конструировать экземпляр обычным способом. Пример, следующий ниже, использует второй подход, чтобы избежать любых сложностей, связанных с особенностями хранилища или используемого фреймворка.

Так же как при сохранении, для решения этой задачи нужно найти точку входа во фреймворк, которая позволит внедрить свою логику парсинга. В NHibernate такой точкой входа является другой метод интерфейса `IUserType`, представленного выше. В листинге 15.29 показано, как получить объект-значение `Name` из хранимого представления.

Листинг 15.29. Парсинг хранимого представления объекта-значения с использованием точки входа во фреймворк

```
// Подготовка NHibernate для применения нестандартной логики отображения
public class NameValueObjectPersister : IUserType
{
    // ..

    public object NullSafeGet(IDataReader rs, string[] names, object owner)
    {
        object storageRepresentation = NHibernateUtil.String.NullSafeGet(
            rs, names[0]
        );
        if (storageRepresentation == null)
        {
            return null;
        }

        // формат хранимого представления: firstName:{X};;surName:{Y}
        var parts = storageRepresentation.ToString().Split(
            new[] { ";;" }, StringSplitOptions.None
        );

        var firstName = parts[0].Split(':')[1];
        var surName = parts[1].Split(':')[1];

        return new Name(firstName, surName);
    }

    // ..
}
```

`IUserType.NullSafeGet()` — это низкоуровневая точка входа в NHibernate, позволяющая вручную воссоздать объект из хранимого представления. Реализация `NullSafeGet()` в листинге 15.29 извлекает `FirstName` и `Surname` из простого текстового представления объекта-значения `Name`, загруженного из хранилища, и на их основе создает новый экземпляр `Name`. Как упоминалось выше, какой бы инструмент ORM вы ни использовали, в нем наверняка найдется аналогичная точка входа, дающая доступ к хранимым представлениям объектов-значений и позволяющая выполнять их парсинг.

В загружаемом пакете примеров для этой главы можно найти полную реализацию `NameValueObjectPersister`, а также модульные тесты, выполняющие подготовку NHibernate перед сохранением и загрузкой данных. На рис. 15.1 показано, что хранится в базе данных после выполнения одного из тестов.

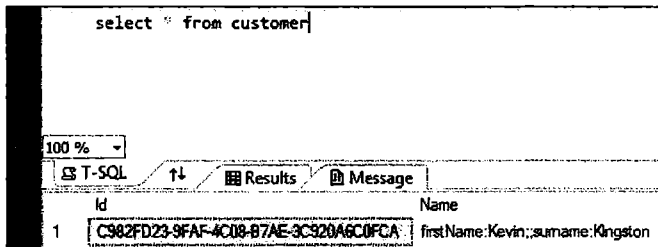


Рис. 15.1. Объект-значение, хранящийся в нестандартном формате

Нормализация в отдельные таблицы

На практике для сохранения объектов-значений широко используется стратегия денормализации, но SQL имеет сильные традиции нормализации. В некоторых компаниях даже могут существовать корпоративные стандарты, прямо требующие выполнять нормализацию. Кроме того, иногда нормализация может оказаться лучшим выбором, например, из соображений производительности и эффективности, особенно если объект-значение имеет представление огромного объема, которое было бы нежелательно загружать всякий раз, когда загружается владеющая им сущность.

В следующем примере вы увидите, как отобразить отношение сущность/объект-значение, где для каждого типа имеется собственная таблица и эти таблицы связаны внешними ключами. Как и в предыдущем примере, для демонстрации будет использоваться фреймворк NHibernate, но сам шаблон можно применить с любой используемой технологией.

Чтобы с помощью NHibernate сохранить объект-значение `Name` в новой таблице, нужно сначала добавить конструктор без аргументов, как показано в листинге 15.30.

Листинг 15.30. Приведение объекта-значения в соответствие с требованиями фреймворка

```
public class Name : ValueObject<Name>
{
    protected Name()
    {
        // Требуется фреймворком NHibernate
    }

    // ..
}
```

Удивительно, как мало дополнительной работы требуется. При использовании FluentNHibernate достаточно просто настроить отображение отношения `Join()`, как показано в листинге 15.31.

Листинг 15.31. Подготовка ORM для сохранения объекта-значения в отдельную таблицу

```
// Класс отображения для FluentNHibernate
public class CustomerNormalizedMap : ClassMap<Customer>
{
    public CustomerNormalizedMap()
    {
        Id(x => x.Id);

        // Создать отдельную таблицу для объекта-значения
        Join("CustomerName", join =>
        {
            join.KeyColumn("Id");
            join.Component(x => x.Name, c =>
            {
                c.Map(x => x.FirstName);
                c.Map(x => x.Surname);
            });
        });
    }
}
```

Используя инструкцию настройки отображения `Join()`, код в листинге 15.31 сообщает пакету `FluentNHibernate`, что для хранения объектов-значений `Name` нужно создать отдельную таблицу `CustomerName`. Инструкция `KeyColumn()` сообщает, что в качестве внешнего ключа должно использоваться поле `ID` в таблице `CustomerName`. Таким образом, когда во время выполнения потребуется загрузить объект `Name`, сущность `Customer` сможет найти его, указав собственное значение `ID`. Какую бы технологию вы ни использовали, в ней наверняка найдутся некоторые соглашения, настройки или точки входа, которые позволят добиться чего-то похожего.

На рис. 15.2 показаны таблицы, созданные фреймворком `NHibernate` в предыдущем примере. Кроме того, на рис. 15.3 можно видеть, как хранятся данные, — обратите внимание, что в качестве внешнего ключа используется значение поля `ID` сущности `Customer`.

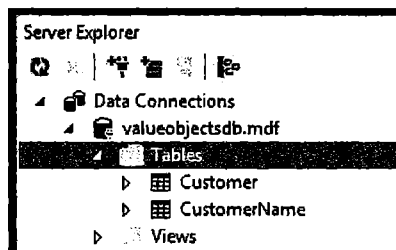


Рис. 15.2. Отдельные таблицы для сущностей `Customer` и объектов-значений `Name`

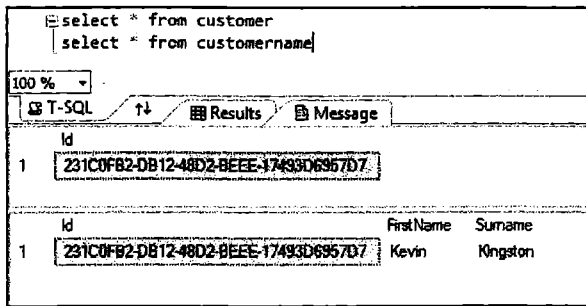


Рис. 15.3. Для соединения таблиц с сущностями Customer и объектами-значениями Name используется внешний ключ таблицы Customer

Ключевые идеи

- Объекты-значения — это компоненты моделей DDD, представляющие описательные характеристики, такие как величины и единицы измерения.
- В связи с тем что объекты-значения не имеют индивидуальности, не следует «захламлять» их сложностями, имеющими отношение к сущностям.
- Элементарные типы, такие как строки и целые числа, рекомендуется оборачивать объектами-значениями, чтобы сделать предметные понятия более ярко выраженными в коде.
- Примерами объектов-значений могут служить: Money, Currency, Name, Height и Color. Однако важно помнить, что объекты-значения в одной предметной области могут оказаться сущностями в другой, и наоборот.
- Объекты-значения не изменяются; их содержимое нельзя изменить.
- Объекты-значения согласованы; они могут иметь множество атрибутов для отражения единственного понятия.
- Объекты-значения могут комбинироваться для создания новых значений без изменения оригиналов.
- Объекты-значения автоматически осуществляют проверку; они никогда не должны оказываться в недопустимом состоянии.
- Существует множество шаблонов моделирования для работы с объектами-значениями, но ничто не мешает вам придумывать свои собственные.
- Содержимое объектов-значений можно сохранять непосредственно в денормализованной форме. Это самый распространенный и простой способ при использовании документоориентированных баз данных и не менее популярный при использовании баз данных SQL.
- Содержимое объектов-значений можно также сохранять в отдельных таблицах или документах. Этот подход обычно применяется с базами данных SQL, но нередко используется для оптимизации при работе с документоориентированными базами данных.

16

Сущности

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в понятие сущностей в DDD
- Выявление сущностей в предметной области
- Различение сущностей и объектов-значений
- Основы реализации сущностей
- Рекомендации по определению предметных моделей с богатыми возможностями за счет создания выразительных сущностей
- Примеры дополнительных шаблонов и принципов проектирования, способных увеличить выразительность сущностей и удобство их сопровождения

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 16 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Представляя свои идеи, специалисты в предметной области часто ссылаются на понятия, характерные для данной предметной области. Например, они могут говорить о конкретном клиенте или конкретном спортивном событии. Эти понятия называют *сущностями* (entities), и, в отличие от объектов-значений, равенство их атрибутов не говорит о том, что это одна и та же сущность.

Выявление сущностей предметной области и их явное моделирование играют важную роль по идейным и техническим соображениям. Если вы определили, что некоторое понятие является сущностью, обратитесь к специалистам в предметной области за выяснением дополнительных деталей, таких как жизненный цикл данной сущности. Важно также уяснить, какие понятия являются сущностями, потому что при их проектировании и реализации применяются различные компромиссы и соображения, в чем вы убедитесь далее в этой главе.

Основной целью этой главы является достижение понимания, что сущности в первую очередь являются понятиями с уникальной индивидуальностью. Несмотря на то что далее будут представлены рекомендации и шаблоны реализации, основанные на многолетнем опыте применения DDD, важно понимать, что многие из описываемых деталей могут меняться с течением времени и появлением новых идей и технологий. Поэтому, понимая принципиальную роль сущностей и имея общие представления об их реализации, вы всегда будете готовы как минимум правильно сформировать свою модель.

Понимание сущностей

В этом разделе предлагается рассмотреть понятие сущности с теоретической точки зрения, а также определить, чем они отличаются от других типов объектов предметной области.

Предметные понятия с индивидуальностью и жизненным циклом

Лучшим способом поиска сущностей в моделируемой предметной области является наблюдение за разговорами специалистов. К примеру, представьте, что вы беседуете с сотрудником туристического агентства. Сотрудник может рассказать вам, что отдыхающие сначала подыскивают гостиницу и только потом бронируют места в ней. Он также может сообщить, что для отдыхающего неприемлемо, если номер будет забронирован в другой гостинице, даже если она будет иметь точно такое же название и схожие условия размещения. Таким образом, сотрудник неявно сообщает, что гостиницы — это сущности, потому что их индивидуальность и уникальность играют важную роль. Без них все гостиницы были бы неотличимы друг от друга, и отдыхающие не смогли бы получить то, что им нужно.

Еще одна деталь, которую можно выяснить у специалиста, — это вид индивидуальности сущности. Во многих случаях индивидуальность сама по себе оказывается важным предметным понятием и может быть смоделирована явно для повышения выразительности предметной модели. Продолжая предыдущий пример, гостиница может иметь широко известное уникальное обозначение, позволяющее разным туристическим агентствам и приложениям делиться такими сведениями о ней, как наличие свободных мест и отзывы отдыхающих. С другой стороны, гостиница может не иметь такого уникального обозначения, и тогда приложению придется сгенерировать его искусственно. Если возникают какие-то неясности, в ваших интересах попросить специалистов рассказать, имеет ли сущность уникальную индивидуальность в предметной области.

Еще одним признаком сущности, помимо индивидуальности, является наличие жизненного цикла. Специалисты в предметной области могут давать вам подсказки, например, в таких фразах: «Заказ принят», «Заказ подтвержден и оплачен», и «Заказ доставлен». Эти фразы указывают, что «нечто» в предметной области имеет жизненный цикл. А чтобы иметь жизненный цикл, «нечто», как правило,

должно обладать индивидуальностью, иначе это «нечто» нельзя будет найти и изменить на разных этапах его жизненного цикла.

Обнаружения сущностей могут произойти в любой момент; не нужно стараться усадить специалистов и вместе с ними пытаться заранее определить все сущности в системе. Некоторые практики DDD начинают с выявления событий, возникающих в предметной области. И уже потом, вместе со специалистами, пытаются понять, какие сущности связаны с каждым событием. Вообще, вы всегда должны находиться в состоянии поиска.

Зависимость от контекста

По общему признанию, порой довольно сложно отличить сущности от других типов объектов предметной области, таких как объекты-значения. Как отмечалось выше, основной отличительной чертой сущностей является индивидуальность — сущность отвечает на вопрос «кто?» или «что?», а не «какой?». И все же порой трудно определить, является некоторое понятие сущностью или объектом-значением, особенно если сущности и объекты-значения зависят от контекста; понятие может быть бесспорной сущностью в одной предметной области и объектом-значением в другой.

Типичным примером такой контекстной зависимости в DDD являются деньги. Возьмем в качестве примера банк. Клиент может положить на свой счет \$100. Спустя некоторое время он снимет \$100 со счета и наверняка получит банкноты и монеты, отличные от тех, что принес в банк, когда клал денежные средства на счет. Однако это отличие не является чем-то существенным, потому что индивидуальность денег в данном случае неважна; клиента интересует лишь сумма. То есть в данном контексте деньги, безусловно, являются объектом-значением. Но в другом контексте, например связанном с производством или отслеживанием перемещений денежных средств, индивидуальность купюр или монет может оказаться важным предметным понятием. То есть в этом случае каждая купюра или монета должна иметь уникальный идентификатор.

Реализация сущностей

Большинство сущностей обладают схожими характеристиками, то есть существуют общие принципы проектирования и приемы реализации, которые вы должны знать, включая идентификацию, применение проверок и делегирование операций. В этом разделе вы познакомитесь с рекомендациями и примерами каждого из основных аспектов проектирования, чтобы при последующих обнаружениях сущностей в своих областях вы могли реализовать их в моделях.

Идентификация

Иногда индивидуальность сущности определяется естественными отличительными признаками, но иногда такие признаки отсутствуют. В первом случае вы долж-

ны в сотрудничестве со специалистами в предметной области выяснить естественные отличительные признаки, а во втором вам придется организовать создание искусственных уникальных идентификаторов в своем приложении, возможно, прибегнув к помощи механизмов хранения.

Естественные ключи

Пытаясь установить индивидуальность сущности, прежде всего необходимо выяснить, имеет ли она уникальный идентификатор в предметной области. Такие идентификаторы называют *естественными ключами*. Вот несколько примеров:

- номера социального страхования;
- название страны;
- номер расчетного счета;
- национальный идентификационный номер;
- код ISBN (для книг).

Вы должны гарантировать неизменность естественного ключа, иначе вам придется предусмотреть массу ссылок в своей системе, указывающих на старый идентификатор. В лучшем случае возникнет сложность, связанная с их своевременным обновлением, в худшем — вы можете забыть обновить несколько ссылок или допустить ошибки, которые приведут к значительным проблемам на уровне предприятия.

После того как вы определите естественный ключ и убедитесь, что он *действительно* является естественным ключом, остается самое простое — присвоить его сущности. Часто для этого достаточно лишь добавить параметр в конструктор, как показано в листинге 16.1.

Листинг 16.1. Передача естественного ключа через параметр конструктора

```
public class Book
{
    public Book(ISBN isbn)
    {
        this.ISBN = isbn;
        this.Id = isbn.Number;
    }

    public string Id { get; private set; }

    public ISBN ISBN { get; private set; }
}
```

Как видно из листинга 16.1, конструктор сущности `Book` имеет параметр `ISBN`, определяющий индивидуальность создаваемого экземпляра. Вы передаете естественный ключ в конструктор, и сущность получает уникальную индивидуальность.

Одна из проблем, связанных с естественным ключом, о которой вы должны знать, заключается в настройке его распознавания в инструменте объектно-реляционно-

го отображения (Object-Relational Mapper, ORM), или технологии доступа к данным. Некоторые фреймворки могут использовать другой способ идентификации сущностей, если явно не сообщить им, что правила идентификации вы будете определять сами. Эта проблема также проявляется при использовании искусственных ключей, о которых рассказывается далее.

Искусственные ключи

Когда сущности предметной области не имеют естественных уникальных идентификаторов, необходимо решить, какого рода идентификатор будет использоваться и как эти идентификаторы будут генерироваться. Типичными примерами таких идентификаторов могут служить: возрастающие последовательности чисел, глобально-уникальные идентификаторы (Globally Unique Identifiers, GUID; их также называют универсально-уникальными идентификаторами — Universally Unique Identifiers, UUID) и строки.

Возрастающие последовательности чисел

Числа, как правило, влекут наименьшие затраты, но их применение означает необходимость иметь глобальный счетчик, где будет храниться последний присвоенный идентификатор. Глобально-уникальные идентификаторы (GUID), с другой стороны, хороши тем, что не имеют такой проблемы — приложение может просто генерировать новые идентификаторы, уникальные по умолчанию, которые тем не менее требуют больше места для хранения. Впрочем, дополнительная память, необходимая для хранения, обходится очень дешево, поэтому использование глобально-уникальных идентификаторов часто является стандартным методом. Строки-идентификаторы обычно имеют определенный формат, включающий, например, хеш-коды, комбинации из нескольких атрибутов или даже отметки времени.

Пример в листинге 16.2 демонстрирует использование возрастающей последовательности чисел, где для хранения последнего присвоенного числового идентификатора используется статическая переменная, а фабричный метод гарантирует присваивание новой сущности следующего числа в качестве идентификатора.

Листинг 16.2. Поддержка глобального счетчика

```
public static class RandomEntityFactory
{
    private static long lastId = 0;

    public static RandomEntity CreateEntity()
    {
        return new RandomEntity(++lastId);
    }
}

public class RandomEntity
{
    public RandomEntity(long Id)
```

```
{  
    this.Id = Id;  
}  
  
public long Id { get; private set; }  
}
```

Если работа приложения будет внезапно прервана из-за ошибки, тогда статическая переменная — `lastId` в листинге 16.2 — потеряет содержащееся в ней значение, а это значит, что почти наверняка старые идентификаторы будут использованы повторно после перезапуска приложения. Для исправления этой проблемы может понадобиться организовать сохранение счетчика. Однако во многих случаях это не самое оптимальное решение из-за дополнительных затрат на чтение и изменение и дополнительной сложности. Некоторые реализации даже требуют применения опасных приемов блокировки в распределенных окружениях или в окружениях с балансировкой нагрузки. Применение возрастающих последовательностей чисел для идентификации в некоторых случаях является лучшим решением, но если создание таких идентификаторов оказывается сложной и опасной задачей, тогда, возможно, применение глобально-уникальных идентификаторов окажется более удачным вариантом.

GUID/UUID

Использование глобально-уникальных идентификаторов может оказаться намного более простым, нежели поддержание глобального счетчика. Во многих случаях они позволяют избавиться от проблем производительности, сложности и синхронизации, как показано в листинге 16.3.

Листинг 16.3. Использование GUID упрощает создание уникальных идентификаторов

```
public static class VehicleFactory  
{  
    public static Vehicle CreateVehicle()  
    {  
        var id = Guid.NewGuid();  
        return new Vehicle(id);  
    }  
}  
  
public class Vehicle  
{  
    public Vehicle(Guid id)  
    {  
        this.Id = id;  
    }  
  
    public Guid Id { get; private set; }  
  
    ...  
}
```

Глобально-уникальные идентификаторы гарантируют свою уникальность, поэтому вне зависимости от того, где будет вызываться метод `GUID.NewGuid()`, он всегда будет возвращать уникальное значение. Соответственно `VehicleFactory.CreateVehicle()` всегда будет создавать объекты `Vehicle` с уникальным `Id` — даже если этот метод одновременно будет вызван на множестве серверов в окружении с балансировкой нагрузки. Этот пример демонстрирует, почему при использовании искусственных ключей глобально-уникальные идентификаторы обычно предпочтительнее.

Глобально-уникальные идентификаторы особенно удобны, когда в браузере выполняется логика, которая должна создать сущность и передать ее нескольким прикладным программным интерфейсам (API) на сервере. Без идентификатора серверные службы не смогли бы узнать, что каждой из них посылается информация об одной и той же сущности. Решить эту проблему можно созданием GUID на стороне клиента в сценарии на JavaScript. Этот процесс иллюстрирует рис. 16.1.

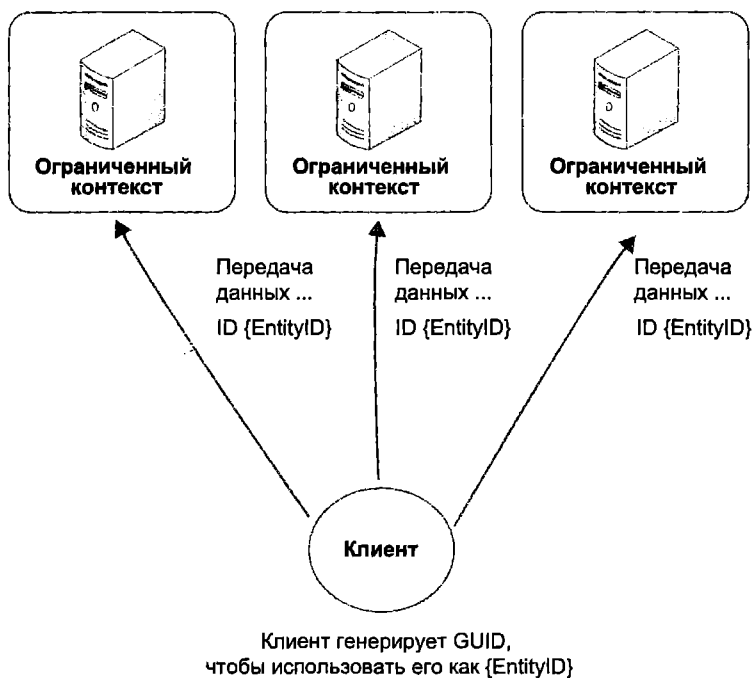


Рис. 16.1. Создание GUID на стороне клиента и передача его нескольким серверным службам

В списке рассылки «DDD/CQRS» (Command Query Responsibility Segregation — разделение ответственности команд и запросов) можно найти длинное обсуждение вопроса создания идентификаторов на стороне клиента. Вам определенно стоит почитать его, если вы планируете использовать этот подход и хотите лучше в нем разобраться (<https://groups.google.com/forum/#!msg/dddccqrs/xYfmh2WwHKk/XW7eauXcKkcJ>).

Строки

Использование строк позволяет определить собственный формат идентификаторов. Существует множество стратегий определения таких форматов. Например, в идентификатор можно включить некоторое состояние сущности в целях диагностики. В листинге 16.4 представлен упрощенный вариант такого подхода, где идентификатором сущности `HolidayBooking` является строка с идентификатором клиента, который забронировал номер, начальной и конечной датой и отметкой времени, когда бронь была подтверждена.

Листинг 16.4. Использование идентификатора с собственным форматом, основанным на атрибутах сущности и отметках времени

```
public class HolidayBooking
{
    public HolidayBooking(int travelerId, DateTime firstNight,
        DateTime lastNight, DateTime booked)
    {
        this.TravelerId = travelerId;
        this.FirstNight = firstNight;
        this.LastNight = lastNight;
        this.Booked = booked;
        this.Id = GenerateId(travelerId, firstNight, lastNight, booked);
    }

    public string Id { get; private set; }

    public int TravelerId { get; private set; }

    public DateTime FirstNight { get; private set; }

    public DateTime LastNight { get; private set; }

    public DateTime Booked { get; private set; }

    private string GenerateId(int travelerId, DateTime firstNight,
        DateTime lastNight, DateTime booked)
    {
        return string.Format(
            "{0}-{1}-{2}-{3}",
            travelerId, ToIdFormat(firstNight),
            ToIdFormat(lastNight), ToIdFormat(booked)
        );
    }

    private string ToIdFormat(DateTime date)
    {
        return date.ToString("yyyy/MM/dd");
    }

    ...
}
```

Конструктор `HotelBooking` в листинге 16.4 вызывает `GenerateId()`, чтобы получить уникальный идентификатор в собственном формате. Это упрощенный пример, но он все же похож на реализации, которые можно встретить в действующих приложениях.

По аналогии с естественными ключами, если вы захотите включить в идентификатор значения атрибутов сущности в целях диагностики, эти значения не должны изменяться впоследствии. Поэтому, определяя свой формат идентификаторов, следует проявлять осторожность и иметь полное представление о характеристиках предметной области.

Идентификаторы, создаваемые базами данных

Часто бывает проще и безопаснее делегировать создание идентификаторов своему хранилищу данных. Большинство баз данных, от реляционных, таких как MS SQL Server, до документоориентированных, таких как RavenDB, имеют встроенную поддержку создания уникальных идентификаторов.

Создание идентификаторов с использованием механизмов баз данных часто осуществляется по сходному шаблону, когда вновь созданная сущность передается в выбранную библиотеку доступа к данным. После успешного завершения следующей транзакции сущность получает уникальный идентификатор. В листинге 16.5 приводится испытательный тест, демонстрирующий реализацию данного решения с использованием NHibernate и SQL Server.

Листинг 16.5. Делегирование создания идентификаторов хранилищу данных посредством ORM

```
[TestClass]
public class DatastoreIdGenerationExample
{
    // реализацию CreateSession() смотрите в загружаемых примерах к этой главе
    ISession session = CreateSession();

    [TestMethod]
    public void Id_is_set_by_datastore_via ORM()
    {
        var entity1 = new IdTestEntity();
        var entity2 = new IdTestEntity();

        // изначально идентификатор не определен
        Assert.AreEqual(0, entity1.Id);
        Assert.AreEqual(0, entity2.Id);

        NHibernateTransaction(session =>
        {
            session.Save(entity1);
            session.Save(entity2);
        });

        // Идентификатор устанавливается посредством NHibernate
```

```
Assert.AreEqual(1, entity1.Id);
Assert.AreEqual(2, entity2.Id);
}

private void NHibernateTransaction(Action<ISession> action)
{
    using (var transaction = session.BeginTransaction())
    {
        action(session);
        transaction.Commit();
    };
}

...
}
```

Пример в листинге 16.5 создает два экземпляра `IdTestEntity` без идентификаторов. Это обстоятельство проверяют две первые инструкции `Assert.AreEqual()`. Далее, в рамках транзакции `NHibernate`, выполняется сохранение обеих сущностей (фактическое сохранение каждой сущности завершается после подтверждения транзакции `NHibernate`). Последние две инструкции `Assert.AreEqual()` проверяют, получили ли сущности ожидаемые идентификаторы. Полный код примера, включая настройки `NHibernate`, можно найти в загружаемом пакете примеров для этой главы.

ПРИМЕЧАНИЕ

Пример в листинге 16.5 использует версии `NHibernate 4.0.0.4000` и `FluentNHibernate 1.4.0.0`. Более новые или старые версии могут иметь отличающиеся API.

Включение логики в объекты-значения и предметные службы

Сохранять сущности сосредоточенными на ответственности за идентификацию очень важно, поскольку это предохраняет их от раздувания — ловушки, в которую легко попасть, если попытаться наделить их множеством связанных функций. Для достижения такой сосредоточенности требуется делегировать выполнение соответствующих функций объектам-значениям и предметным службам. Примеры такого делегирования уже демонстрировались в предыдущей главе, когда мы обсуждали возможность комбинирования, сравнения и автоматической проверки в объектах-значениях (мы выносили логику за пределы сущностей, использующих их). Аналогично, если заглянуть в главу 17 «Службы предметной области», можно найти множество случаев, когда предметные операции, которые не имеют состояния и которые, на первый взгляд, должны выполняться сущностями, в действительности можно переложить на предметные службы.

Для демонстрации преимуществ включения логики в объекты-значения в листинге 16.6 представлена измененная версия сущности `HolidayBooking`, которая

прежде приводилась в листинге 16.4. Новая версия реализует ключевые правила предметной области с использованием единого языка. Первое из них требует, чтобы первая ночь отпуска предшествовала последней. Второе требует, чтобы номер в гостинице бронировался не менее чем на три ночи. Проверка соответствия обоим этим требованиям перекладывается на объект-значение Stay.

Листинг 16.6. Включение логики поведения сущности в объект-значение

```
public class HolidayBooking
{
    public HolidayBooking(int travelerId, Stay stay, DateTime booked)
    {
        this.TravelerId = travelerId;
        this.Stay = stay;
        this.Booked = booked;
        this.Id = GenerateId(
            travelerId, stay.FirstNight, stay.LastNight, booked
        );
    }

    public string Id { get; private set; }

    public int TravelerId { get; private set; }

    public Stay Stay { get; private set; }

    public DateTime Booked { get; private set; }

    private string GenerateId(int travelerId, DateTime firstNight,
        DateTime lastNight, DateTime booked)
    {
        return string.Format(
            "{0}-{1}-{2}-{3}",
            travelerId, ToIdFormat(firstNight), ToIdFormat(lastNight),
            ToIdFormat(booked)
        );
    }

    private string ToIdFormat(DateTime date)
    {
        return date.ToString("yyyy/MM/dd");
    }
}

public class Stay
{
    public Stay(DateTime firstNight, DateTime lastNight)
    {
        if (firstNight > lastNight)
            throw new FirstNightOfStayCannotBeAfterLastNight();
    }
}
```



```
        if (DoesNotMeetMinimumStayDuration(firstNight, LastNight))
            throw new StayDoesNotMeetMinimumDuration();

        this.FirstNight = firstNight;
        this.LastNight = lastNight;
    }

    public DateTime FirstNight { get; private set; }

    public DateTime LastNight { get; private set; }

    private bool DoesNotMeetMinimumStayDuration(DateTime firstNight,
                                                DateTime lastNight)
    {
        return (lastNight - firstNight) < TimeSpan.FromDays(3);
    }
}
```

В листинге 16.4 демонстрируется начальная версия сущности `HolidayBooking`, сосредоточенная исключительно на реализации индивидуальности. Ее легко можно было бы наделить новыми функциями, включив их непосредственно в этот класс, не создавая объекта-значения `Stay`, представленного в листинге 16.6. Однако в этом случае ухудшилась бы выразительность сущности из-за смешивания логики, имеющей отношение к индивидуальности, с логикой, связанной с периодом пребывания (к которому применяется бронирование).

Может показаться, что класс `Stay` очень мал и реализованную в нем логику без всякого ущерба можно было бы включить непосредственно в `HolidayBooking`. Но представьте другие аспекты бронирования, такие как отдельно оплачиваемые переезды или перелеты. Если ответственность за все это включить в сущность `HolidayBooking`, она завуалирует и запутает предметные понятия. Всякий раз, добавляя новую функцию в сущность, следует подумать, нельзя ли реализовать ее в виде объекта-значения для улучшения ясности предметной области.

ПРИМЕЧАНИЕ

Вообще, следует избегать наделения любых классов более чем одной ответственностью, потому что в противном случае их будет труднее сопровождать, труднее тестировать и они превратятся в ту часть программного кода, необходимость работать с которой не будет вызывать энтузиазма. Соблюдение принципа единственной ответственности (Single Responsibility Principle, SRP) — отличный совет, который можно дать тем, кто занимается разработкой программного обеспечения в целом и сущностей в частности. Например, логику, заключенную в объекте-значении, легче изменить и протестировать просто потому, что при этом не затрагиваются другие обязанности.

В стремлении сохранить компактность сущностей за счет включения логики в объекты-значения следует учитывать такой важный аспект, как глубина иерархии объектов. В листинге 16.6 свойство `Stay` является общедоступным, соответственно, общедоступными являются и его свойства `FirstNight` и `LastNight`. Это

позволяет, например, обратиться к `HotelBooking.Stay.FirstNight`. Глубина вложенности в три уровня в данном случае не кажется неоправданно большой, но все же следует следить за количеством экспортируемых уровней иерархии объектов, потому что клиенты будут зависеть от этого. В данном сценарии возможная реорганизация свойств `FirstNight` и `LastNight` может быть осложнена из-за тесной зависимости других частей системы от текущей организации кода. В каждом конкретном случае вам придется оценивать, до какой глубины в иерархии объектов выгодно открыть общий доступ.

ПРИМЕЧАНИЕ

Спор о необходимости предоставления клиентам доступа к свойству `FirstNight` объекта-значения `Stay` тесно связан с принципом объектно-ориентированного программирования под названием «Закон Деметры» (The Law of Demeter, LoD). Для обсуждения закона Деметры Фил Хаак (Phil Haack), прежде работавший в Microsoft, опубликовал в своем блоге статью, которая отражает мнение большинства по данному вопросу (<http://haacked.com/archive/2009/07/14/law-of-demeter-dot-counting.aspx/>).

Проверка и соблюдение правил

В дополнение к индивидуальности, к реализациям сущностей предъявляется еще одно основное требование — они должны осуществлять автоматическую проверку и всегда находиться в допустимом состоянии. Это похоже на автоматическую проверку в объектах-значениях, с той лишь разницей, что в сущностях такая проверка в большой степени зависит от контекста из-за наличия у сущностей жизненного цикла. Например, сущность `FlightBooking` может позволять изменять свое свойство `DepartureDate`, пребывая в состоянии ожидания подтверждения из авиакомпании. Однако, получив подтверждение, правила валидации запрещают изменять `DepartureDate` в соответствии с политиками компании. В листинге 16.7 приводится реализация сущности `FlightBooking`, демонстрирующая упомянутые контекстно-зависимые правила проверки.

Листинг 16.7. Контекстно-зависимая проверка и проверка в конструкторе

```
public class FlightBooking
{
    private bool confirmed = false;

    public FlightBooking(Guid id, DateTime departureDate, Guid customerId)
    {
        if (id == null)
            throw new IdMissing();

        if (departureDate == null)
            throw new DepartureDateMissing();

        if (customerId == null)
            throw new CustomerIdMissing();

        this.Id = id;
```

```
        this.DepartureDate = departureDate;
        this.CustomerId = customerId;
    }

    public Guid Id { get; private set; }

    public DateTime DepartureDate { get; private set; }

    public Guid CustomerId { get; private set; }

    public void Reschedule(DateTime newDeparture)
    {
        if (confirmed) throw new RescheduleRejected();
        this.DepartureDate = newDeparture;
    }

    public void Confirm()
    {
        this.confirmed = true;
    }
}
```

Когда вызывается метод `Reschedule()` с измененной датой вылета, сущность `FlightBooking`, представленная в листинге 16.7, возбуждает исключение `RescheduleRejected`. Но этого не происходит, если бронь на билет еще не была подтверждена. Таким образом, данная проверка зависит от контекста, потому что применяется только в определенных сценариях. Может показаться, что это противоречит заявлению: «сущности всегда находятся в допустимом состоянии». Однако это не так, в данном случае фразу «всегда находятся в допустимом состоянии» следует толковать как «всегда находятся в допустимом состоянии для данного контекста».

Как можно заметить в листинге 16.7, конструктор проверяет каждый свой аргумент. Это еще одно важное правило, гарантирующее, что сущности всегда будут находиться в допустимом состоянии. Это также предотвратит проникновение проблем в предметную область и не даст ей оказаться в недопустимом состоянии. В данном примере в реализации конструктора используются специфические типы исключений, но ничто не мешает вам реализовать проверку иначе, по своему усмотрению. В общем случае чем выразительнее, тем лучше.

ВНИМАНИЕ

В некоторых примерах в этой главе проверка аргументов опущена, чтобы сделать эти примеры более ясными и прозрачными. Однако чтобы отказаться от проверки аргументов конструкторов в действующих приложениях, нужно иметь весьма веские причины, потому что такое решение может повлечь серьезные неприятности.

Сущности также несут ответственность за реализацию более фундаментальной формы проверки: инвариантов. Инварианты — это факты о сущностях. В целях более точного представления моделируемой сущности они требуют, чтобы значения некоторых атрибутов находились в определенных диапазонах.

Продолжая пример с гостиницами, для определения инвариантов¹ спросите себя: «Что делает гостиницу гостиницей?» или «Что для гостиницы означает быть гостиницей?». С точки зрения индустрии туризма и отдыха гостиница — это здание с комнатами, которые могут снимать путешественники и отдыхающие на определенное время. Следовательно, если в здании нет комнат, оно не может быть гостиницей. В листинге 16.8 показано, как реализуется инвариант «чтобы быть гостиницей, нужны комнаты».

Листинг 16.8. Реализация инвариантов в сущности

```
public class Hotel
{
    public Hotel(Guid id, HotelAvailability initialAvailability,
                HotelRoomSummary rooms)
    {
        ...

        EnforceInvariants(rooms);
        this.Id = id;
        this.Availability = initialAvailability;
        this.Rooms = rooms;
    }

    private void EnforceInvariants(HotelRoomSummary rooms)
    {
        if (rooms.NumberOfSingleRooms < 1 &&
            rooms.NumberOfDoubleRooms < 1 &&
            rooms.NumberOfFamilyRooms < 1)
            throw new HotelsMustHaveRooms();
    }

    public Guid Id { get; private set; }

    public HotelAvailability Availability { get; private set; }

    public HotelRoomSummary Rooms { get; private set; }

}
```

Реализация инвариантов не должна быть особенно сложной и в идеале должна быть явной. Это подтверждается методом `EnforceInvariants()` в листинге 16.8. Прежде чем начнется создание экземпляра `Hotel`, конструктор вызывает этот метод и убеждается, что аргументы удовлетворяют основным требованиям к гостиницам. В данном случае под этим понимается наличие в гостинице хотя бы одной комнаты. Если это условие не выполняется, возбуждается исключение `HotelsMustHaveRooms`, не оставляя никаких сомнений по поводу инварианта и никакой возможности обойти его.

¹ Как отмечалось в главе 14, инварианты (invariants) — это правила, обеспечивающие согласованность в предметной модели. — *Примеч. пер.*

Как уже упоминалось, реализации проверок допустимости и инвариантов похожи и внутренние, и внешние. Доказательства этого можно увидеть, если сравнить листинги 16.7 и 16.8. Однако есть одно небольшое, но важное отличие: инвариант в листинге 16.8 проверяется всегда, вне зависимости от контекста. Мелкие отличия, подобные этому, очень важны, так как при знакомстве с предметной областью вы должны научиться отличать контекстные правила проверки от инвариантов, чтобы до конца понять предметную область и смоделировать ее.

Далее в этой главе вы увидите, что проверки можно выносить за пределы сущностей с использованием спецификаций. Вы также увидите, что когда сущность имеет множество возможных состояний, может потребоваться смоделировать их явно, если вы захотите применить шаблон «Состояние» (state).

Сосредоточьтесь на поведении, а не на данных

По общему мнению многих практиков DDD, сущности должны быть ориентированы на реализацию поведения. То есть сущности должны экспортировать методы с выразительными именами, сообщающими особенности поведения предметных понятий, а не их состояние. В более широком смысле это близко соответствует принципу объектно-ориентированного программирования «говори, а не спрашивай».

Сосредоточенность на поведении сущности в DDD играет важную роль, потому что такой подход делает предметную модель более выразительной. Кроме того, закрывая прямой доступ к состоянию сущности, мы получаем состояние, которым может управлять только сам экземпляр сущности. Это означает, что любые функции, изменяющие состояние сущности, должны принадлежать самой сущности. Это особенно желательно, так как не позволяет реализовать логику, принадлежащую сущности, в неправильном месте.

Занимаясь реализацией сущностей, сосредоточенных на поведении, нужно с осторожностью подходить к экспортированию методов чтения (getters) и особенно к экспортированию методов записи (setters). Доступность методов записи увеличивает риск изменения состояния сущности другими частями системы такими способами, которые не объясняют, зачем эти изменения выполняются, скрывая предметное понятие или причину, обусловившую изменение, как демонстрирует следующий пример.

При моделировании предметной области, где требуется отслеживать результаты футбольных матчей, вы можете задуматься о реализации сущности `SoccerCupMatch`, как показано в листинге 16.9, состояние которой открыто для общего доступа. Однако прежде чем действительно принять такое решение и реализовать архитектуру, ориентированную на состояние, нужно очень хорошо всё предусмотреть.

Листинг 16.9. Сущность, открывающая доступ к своему состоянию

```
public class SoccerCupMatch
{
    public SoccerCupMatch(Guid id, Scores team1Scores, Scores team2Scores)
    {
        if (id == null)
```

```

        throw new ArgumentNullException(
            "Soccer cup match ID cannot be null"
        );

        if (team1Scores == null)
            throw new ArgumentNullException(
                "Team 1 scores cannot be null"
            );

        if (team2Scores == null)
            throw new ArgumentNullException(
                "Team 2 scores cannot be null"
            );

        this.ID = id;
        this.Team1Scores = team1Scores;
        this.Team2Scores = team2Scores;
    }

    public Guid ID { get; private set; }

    public Scores Team1Scores { get; set; }

    public Scores Team2Scores { get; set; }
}

```

Экспортируя состояние сущности для чтения и изменения извне, как в примере с сущностью `SoccerCupMatch` в листинге 16.9, вы открываете возможность реализации поведения, которое должно принадлежать сущности, где-то еще, что ухудшает ясность и выразительность. Например, в футболе есть интересное правило определения победителя по количеству мячей, забитых в гостях. Две команды играют в турнире друг с другом дважды, каждая из них играет один раз на своем стадионе и один раз в гостях. Если в общем счете получается ничья, тогда победителем становится команда, забившая большее число голов в гостях. Например, если первая игра на стадионе команды А окончилась со счетом 1–0 в пользу команды А, а вторая игра, на стадионе команды Б, — со счетом 2–1 в пользу команды Б, тогда общий счет получается 2–2. Однако команда А забила 1 мяч в гостях, а команда Б — 0 мячей. Поэтому победителем будет считаться команда А.

Теперь взгляните еще раз на реализацию сущности `SoccerCupMatch` в листинге 16.9. Как видите, она экспортирует только свое состояние — счет. Клиенты этой сущности могут увидеть счет 2–2 и по ошибке посчитать, что результатом является ничья или что команда Б победила. Теоретически сущность может иметь множество клиентов, каждый из которых реализует свою, отличную от других, логику определения победителя. Такая несогласованность может привести к странному и ошибочному поведению в предметной модели. Поэтому сущность `SoccerCupMatch` должна закрыть доступ к своему состоянию и экспортировать методы определения победителя, как показано в улучшенной версии в листинге 16.10.

Листинг 16.10. Сущность, экспортирующая поведение вместо состояния

```
public class SoccerCupMatch
{
    public SoccerCupMatch(Guid id, Scores team1Scores, Scores team2Scores)
    {
        ...
    }

    public Guid ID { get; private set; }

    private Scores Team1Scores { get; set; }

    private Scores Team2Scores { get; set; }

    public Scores WinningTeamScores
    {
        get
        {
            if (Team1Scores.TotalScore > Team2Scores.TotalScore)
                return Team1Scores;

            if (Team2Scores.TotalScore > Team1Scores.TotalScore)
                return Team2Scores;

            var awayGoalsWinner = FindWinnerUsingAwayGoalsRule();
            if (awayGoalsWinner == null)
                return FindWinnerOfPenaltyShootout();
            else
                return awayGoalsWinner;
        }
    }

    private Scores FindWinnerUsingAwayGoalsRule()
    {
        if (Team1Scores.AwayLegGoals > Team2Scores.AwayLegGoals)
            return Team1Scores;

        if (Team2Scores.AwayLegGoals > Team1Scores.AwayLegGoals)
            return Team2Scores;

        // Число мячей, забитых в гостях, одинаково,
        // поэтому нет возможности определить победителя
        return null;
    }

    private Scores FindWinnerOfPenaltyShootout()
    {
        if (Team1Scores.ShootoutScore > Team2Scores.ShootoutScore)
            return Team1Scores;

        if (Team2Scores.ShootoutScore > Team1Scores.ShootoutScore)
```

```
        return Team2Scores;

        throw new ThereWasNoPenaltyShootout();
    }
}
```

Даже не зная правил игры в футбол, взглянув на реализацию `SoccerCupMatch` в листинге 16.10, можно понять суть правила определения победителя по числу мячей, забитых в гостях. Также важно отметить, что теперь, благодаря закрытому состоянию сущности, экспортируется только ее поведение. Клиенты `SoccerCupMatch` могут взаимодействовать с экземплярами `SoccerCupMatch`, только используя предметную терминологию, а не просто обращаясь к изменяемому состоянию. Это большой шаг к созданию функциональных и выразительных предметных моделей.

Однако порой возникают ситуации, когда, казалось бы, нет никакого выбора, кроме как экспортировать методы чтения и записи. Далее в этой главе вы узнаете, как шаблон «Хранитель» (*memento*) может ослабить необходимость экспортирования методов чтения и записи.

ПРИМЕЧАНИЕ

Полное описание правила определения победителя по числу мячей, забитых в гостях, можно найти в Википедии: https://ru.wikipedia.org/wiki/Правило_гола,_забитого_на_чужом_поле.

Избегайте ошибки «моделирования реального мира»

Даже при том что точное отражение поведения предметной области является основой DDD, многие начинающие применять эту философию по ошибке стремятся смоделировать всё и вся. По неопытности они считают, что методология DDD связана с моделированием реального мира, и, как следствие, стремятся смоделировать всё, что только можно увидеть в реальности, а не то, что действительно необходимо в приложениях. Это весьма распространенная ошибка, и совершенно очевидно, что она обусловлена самыми лучшими намерениями. К сожалению, из-за этой ошибки увеличивается число понятий и сложностей в предметной модели. Это также ведет к путанице, если оказывается, что какие-то части программного кода вообще никогда не используются.

Как только вы поймете, что предметная модель не должна быть полномасштабной моделью фактической предметной области, вы уже не допустите этой ошибки, а если и допустите, то сможете быстро найти и устранить ее. Однако существует схожая проблема, обнаружить которую зачастую гораздо труднее; иногда сущности формируются под давлением требований, предъявляемых пользовательским интерфейсом. Часто, во избежание захламления интерфейса сущности постоянными обязанностями, бывает более целесообразно добавит в представление модели или в объект передачи данных (*Data Transfer Object, DTO*) переходную логику, связанную с пользовательским интерфейсом.

ПРИМЕЧАНИЕ

В главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования» даются рекомендации по передаче приватного состояния через слой служб в пользовательский интерфейс. Хорошим примером такого подхода может служить демонстрация шаблона «Посетитель» (visitor). Также с успехом можно использовать шаблон «Хранитель», который демонстрируется далее в этой главе.

Явным признаком проблемы является проникновение технических понятий в сущность. К числу типичных примеров можно отнести включение атрибутов внедрения зависимостей, атрибутов проверки или реализацию функциональности, присущей ORM, такой как отложенная загрузка данных. Считается, что все это также не должно захламлять сущности, так как ухудшается выразительность предметной модели. Обычно лучше избегать подобных «загрязнений» сущности, если это возможно. По общему признанию, иногда разработчики действительно оказываются в ситуации, когда приходится балансировать на грани между желанием получить чистую и ясную предметную модель и возможностью достичь цели самым простым способом. В таких ситуациях можно стиснуть зубы и пойти на ухудшение выразительности модели, но прежде вы должны убедиться, что сделали всё, что в ваших силах.

Проектирование для распределенных окружений

В последние годы распределенные системы постепенно становятся все более обычными. Как следствие, возникают новые подходы к проектированию предметных моделей и, соответственно, сущностей. Однако многие практики DDD в подавляющем большинстве случаев настоятельно не рекомендуют создавать распределенные сущности. Фактически это означает, что сущность должна быть ограничена рамками одного класса, одного ограниченного контекста, одной предметной модели. Понять суть этой рекомендации вам поможет пример типичной сущности Customer, представленной в листинге 16.11.

Листинг 16.11. Сущность, логически простирающаяся на несколько ограниченных контекстов

```
public class Customer
{
    public Customer(Guid id, AddressBook addresses, Orders orderHistory,
        PaymentDetails paymentDetails, LoyaltySummary loyalty)
    {
    }

    public Guid Id { get; private set; }

    public AddressBook Addresses { get; private set; }

    public Orders OrderHistory { get; private set; }

    public PaymentDetails PaymentDetails { get; private set; }

    public LoyaltySummary Loyalty { get; private set; }
}
```

Адреса, персональные данные, история заказов, данные платежей и уровень лояльности — все это представлено в сущности *Customer* в листинге 16.11. В монолитном приложении определено могут быть причины для такой организации. Но в распределенной системе адреса, заказы, данные платежей и информация о лояльности могут находиться в разных ограниченных контекстах. В таком случае, чтобы загрузить такую сущность из хранилища, может потребоваться выполнить несколько запросов к разным базам данных в разных ограниченных контекстах. Как рассказывалось в главах с 11-й по 13-ю, тесная связь между распределенными компонентами отрицательно сказывается на масштабируемости и надежности и влечет другие проблемы, обусловленные необходимостью выполнения сетевых операций.

С точки зрения распределенной организации сущность *Customer* в листинге 16.11 фактически представляет множество разных понятий из разных ограниченных контекстов. В листинге 16.12 показано, как реорганизовать представление информации о лояльности, истории заказов и данных платежей для сущности *Customer*, чтобы обеспечить возможность ее использования в распределенных ограниченных контекстах.

Листинг 16.12. Сущности с поддержкой размещения ограниченных контекстов в распределенном окружении

```
namespace MarketingBoundedContext
{
    public class Loyalty
    {
        ...

        public Guid CustomerId { get; private set; }

        public LoyaltySummary Loyalty { get; private set; }

    }
}

namespace AccountsBoundedContext
{
    public class OrderHistory
    {
        ...

        public Guid CustomerId { get; private set; }

        public Orders Orders { get; private set; }

        ...
    }
}

namespace BillingBoundedContext
{
    public class PaymentDetails
```

```
{  
    ...  
  
    public Guid CustomerId { get; private set; }  
  
    public CardDetails Default { get; private set; }  
  
    public CardDetails Alternate { get; private set; }  
  
}
```

Каждая из сущностей в листинге 16.12 содержит часть функциональности, заимствованную из непомерно раздутой сущности `Customer`, представленной в листинге 16.11. Все эти сущности имеют поле `CustomerId`, объединяющее информацию, которая в них содержится, несмотря на то что они находятся в разных ограниченных контекстах. Такое решение наверняка окажется менее проблематичным в распределенной системе. И, как дополнительная выгода, включение ограниченных контекстов будет лучше соответствовать предметной области.

Даже до того, как философия DDD стала применяться к распределенным окружениям, при проектировании систем всегда рекомендовалось учитывать возможность их дробления на ограниченные контексты и распределение обязанностей, как в случае с сущностью `Customer` из листинга 16.11. Поэтому рекомендация, упомянутая в данном разделе применительно к распределенным окружениям, в действительности справедлива для любых реализаций, спроектированных с применением философии DDD или способных извлечь дополнительные преимущества от наличия множества ограниченных контекстов.

Основные принципы и шаблоны моделирования сущностей

В этом разделе будет представлено несколько принципов и шаблонов, которые могут улучшить выразительность сущностей и упростить их сопровождение в будущем. Однако это не единственно возможные шаблоны. В действительности цель данного раздела — показать, что при реализации сущностей всегда есть место творчеству — при условии учета фундаментальных положений, о которых рассказывалось в предыдущем разделе.

Реализуйте проверки и инварианты с помощью спецификаций

Спецификации (specifications) — это маленькие, узкоспециализированные классы, напоминающие правила. Преимущество использования спецификаций для организации проверок и инвариантов заключается в улучшенной выразительности за

счет реализации каждого единственного понятия в виде отдельного класса, а также в простоте тестирования благодаря неизменяемости и отсутствию побочных эффектов. Спецификации также могут служить примером вынесения поведения за пределы сущностей в объекты других типов, как рекомендовалось ранее в этой главе.

В листинге 16.13 приводится выдержка из реализации измененной версии сущности `FlightBooking`, которая упоминалась ранее. Эта версия использует спецификацию для реализации правила перепланирования.

Листинг 16.13. Использование простых спецификаций для выполнения проверок

```
public class FlightBooking
{
    private NoDepartureReschedulingAfterBookingConfirmation spec =
        new NoDepartureReschedulingAfterBookingConfirmation();

    ...

    public void Reschedule(DateTime newDeparture)
    {
        if (!spec.IsSatisfiedBy(this)) throw new RescheduleRejected();

        this.DepartureDate = newDeparture;
    }

    ...
}
```

Основная логика проверки возможности переноса вылета на другую дату теперь заключена в отдельном классе с именем `NoDepartureReschedulingAfterBookingConfirmation`, которое отлично описывает применяемое правило. Реализация этого класса приводится в листинге 16.14 наряду с обобщенным интерфейсом `ISpecification`, который он реализует.

Листинг 16.14. Реализация спецификации и обобщенный интерфейс

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T Entity);
}

public class NoDepartureReschedulingAfterBookingConfirmation :
    ISpecification<FlightBooking>
{
    public bool IsSatisfiedBy(FlightBooking booking)
    {
        return !booking.Confirmed;
    }
}
```

Спецификация, использующая эту разновидность шаблона, имеет единственный метод `IsSatisfiedBy()`, как показано в листинге 16.14. `IsSatisfiedBy()` должен

возвращать `false`, если моделируемое правило не может быть применено к переданному ему объекту.

У кого-то из вас наверняка появится вопрос: окупятся ли трудозатраты, связанные с созданием отдельных спецификаций и использованием обобщенного интерфейса. Конечно, ситуация выглядит неоднозначно — требуется создать отдельные спецификации и определить интерфейс. Однако этот шаблон имеет другой ключевой аспект, позволяющий извлечь выгоду из абстракции `ISpecification` и способный убедить вас в его полезности: возможность объединения спецификаций, как показано в листинге 16.15.

Листинг 16.15. Объединение спецификаций

```
new OrSpecification<FlightBooking>(  
    new FrequentFlyersCanRescheduleAfterBookingConfirmation(),  
    new NoDepartureReschedulingAfterBookingConfirmation()  
);
```

Некоторые авиакомпании, выполняющие частые рейсы в одном направлении, предоставляют небольшие льготы, например возможность перепланирования за короткий срок до вылета. Если понадобится изменить правило «запретить перенос даты вылета после подтверждения», чтобы оно не применялось к выбранным клиентам, это изменение можно смоделировать с помощью `OrSpecification`, как показано в листинге 16.15. `OrSpecification` сначала попытается применить первую спецификацию, и если попытка потерпит неудачу, тогда будет выполнена попытка применить другую спецификацию. Теперь листинг 16.15 должен быть вам понятен; `OrSpecification` сначала пытается применить спецификацию `FrequentFlyersCanRescheduleAfterBookingConfirmation`, и только если она потерпит неудачу, вызывается спецификация `NoDepartureReschedulingAfterBookingConfirmation` в соответствии с политиками компании в данной предметной области.

Реализацию `OrSpecification` можно увидеть в листинге 16.16.

Листинг 16.16. Обобщенная спецификация для объединения других спецификаций

```
public class OrSpecification<T> : ISpecification<T>  
{  
    private ISpecification<T> first;  
    private ISpecification<T> second;  
  
    public OrSpecification(ISpecification<T> first, ISpecification<T> second)  
    {  
        this.first = first;  
        this.second = second;  
    }  
  
    public bool IsSatisfiedBy(T entity)  
    {  
        return first.IsSatisfiedBy(entity) || second.IsSatisfiedBy(entity);  
    }  
}
```

Опираясь на фундамент `ISpecification`, можно организовать объединение спецификаций другими способами. Например, можно определить спецификацию `AndSpecification`, требующую, чтобы обе переданные ей спецификации вернули `true`. Кроме того, можно создать базовый класс или применить шаблон «Строитель» (builder) для гибкого объединения спецификаций, как предлагается в листинге 16.17.

Листинг 16.17. Гибкое объединение спецификаций

```
var ffSpec = new FrequentFlyersCanRescheduleAfterBookingConfirmation();
var ndSpec = new NoDepartureReschedulingAfterBookingConfirmation();

var spec = ffSpec.Or(ndSpec).Or( ... );
```

Избегайте шаблона «Состояние»; используйте явное моделирование

Во многих предметных областях есть сущности, жизненный цикл которых включает несколько стадий, или состояний. В каждом состоянии обычно допустимым оказывается только некоторое подмножество функциональности сущности. Например, процесс заказа еды на дом может включать следующие состояния: «в очереди для передачи на кухню», «подготовка», «в процессе приготовления» и «передан для доставки». Очевидно, что когда заказ передан для доставки, его нельзя поместить обратно в печь, или, когда заказ находится в печи, уже нельзя начать его подготовку, потому что это было выполнено ранее.

Из-за наличия жизненного цикла иногда возникает соблазн реализовать сущность с использованием шаблона «Состояние». Однако некоторые практики DDD настоятельно рекомендуют не использовать этот шаблон для сущностей и предлагают реализовать отдельные классы для каждого состояния.

В листинге 16.18 представлена реализация сценария заказа еды на дом с применением шаблона «Состояние», как только что было описано, в виде сущности `OnlineTakeawayOrder` с интерфейсами для каждого состояния, а также реализация одного из состояний для примера.

Листинг 16.18. Реализация сущности с применением шаблона «Состояние»

```
public class OnlineTakeawayOrder
{
    // состояние, которому делегируется выполнение операций
    private IOnlineTakeawayOrderState state;

    public OnlineTakeawayOrder(Guid id, Address address)
    {
        ...

        this.Id = id;
        this.Address = address;
        this.state = new InKitchenQueue(this);
    }
}
```

```
}

public Guid Id { get; private set; }

public Address Address { get; private set; }

public void Cook()
{
    state = state.Cook();
}

public void TakeOutOfOven()
{
    state = state.TakeOutOfOven();
}

public void Package()
{
    state = state.Package();
}

public void Deliver()
{
    state = state.Deliver();
}
}

// Этот интерфейс реализует каждое состояние
public interface IOnlineTakeawayOrderState
{
    IOnlineTakeawayOrderState Cook();

    IOnlineTakeawayOrderState TakeOutOfOven();

    IOnlineTakeawayOrderState Package();

    IOnlineTakeawayOrderState Deliver();
}

public class InKitchenQueue : IOnlineTakeawayOrderState
{
    private OnlineTakeawayOrder order;

    public InKitchenQueue(OnlineTakeawayOrder order)
    {
        this.order = order;
    }

    public IOnlineTakeawayOrderState Cook()
    {
        // соответствующая реализация состояния
    }
}
```

```

...
return new InOven(order);
}

// все методы ниже неприменимы к этому состоянию
public IOnlineTakeawayOrderState TakeOutOfOven()
{
    throw new ActionNotPermittedInThisState();
}

public IOnlineTakeawayOrderState Package()
{
    throw new ActionNotPermittedInThisState();
}

public IOnlineTakeawayOrderState Deliver()
{
    throw new ActionNotPermittedInThisState();
}
}

```

Листинг 16.18 подчеркивает самый большой недостаток шаблона «Состояние»: его применение может привести к появлению массы типового кода и нереализованных методов. Эти проблемы наглядно демонстрирует класс `InKitchenQueue`, который соответствует одному из пяти возможных состояний в этом упрощенном сценарии. Каждый класс-состояние реализует единственный метод, имеющий определенное назначение. Методы, вызовы которых недопустимы в том или ином состоянии, возбуждают исключение `ActionNotPermittedInState`. Несмотря на то что этот пример в какой-то мере является крайним случаем, где реализация каждого состояния имеет только один метод, выполняющий фактические функции, многие реализации шаблона «Состояние» в действительности содержат еще больше бесполезного программного кода.

Большее значение для DDD имеет тот факт, что шаблон «Состояние» оказывается менее четким. Допуская присутствие методов, которые не должны вызываться, шаблон «Состояние» не делает правила предметной области ясными и четкими. Некоторые практики DDD утверждают, что если нет возможности упаковать заказ, еще находящийся в печи, это ограничение должно быть четко выражено на уровне системы типов. Этот идеал достижим, если для каждого состояния определить свою сущность, включающую только операции, которые поддерживаются в данном состоянии. В листинге 16.19 показан результат применения этой философии к сценарию заказа еды на дом.

Листинг 16.19. Замена шаблона «Состояние» явным моделированием состояний

```

// эти сущности, все вместе, являются заменой
// единственной сущности OnlineTakeawayOrder
public class InKitchenOnlineTakeawayOrder
{
    public InKitchenOnlineTakeawayOrder(Guid id, Address address)

```



```
{
    ...

    this.Id = id;
    this.Address = address;
}

public Guid Id { get; private set; }

public Address Address { get; private set; }

// содержит только необходимые методы
// возвращает новое состояние, чтобы известить о нем клиента
public InOvenOnlineTakeawayOrder Cook()
{
    ...

    return new InOvenOnlineTakeawayOrder(this.Id, this.Address);
}
}

public class InOvenOnlineTakeawayOrder
{
    public InOvenOnlineTakeawayOrder(Guid id, Address address)
    {
        ...

        this.Id = id;
        this.Address = address;
    }

    public Guid Id { get; private set; }

    public Address Address { get; private set; }

    public CookedOnlineTakeawayOrder TakeOutOfOven()
    {
        ...

        return new CookedOnlineTakeawayOrder(this.Id, this.Address);
    }
}

// другие три состояния можно найти в пакете примеров для этой главы
```

Листинг 16.19 демонстрирует две новые сущности, которые делают излишними определение и реализацию интерфейса `IOneOnlineTakeawayOrderState`. Как можно заметить, избыточный типовой код полностью исчез. Однако истинная выгода состоит в том, что теперь правила предметной области выражены в программном коде более четко; исчезла единственная сущность `OnlineTakeawayOrder`, которая передавалась через всю предметную модель и позволяла вызывать недопустимые методы. Вместо этого теперь модель должна явно указать, какое состояние ей необходимо для дальнейшей работы, в результате чего предметные правила становятся более выразительными и гарантируются системой типов.

Избегайте использования методов чтения/записи с шаблоном «Хранитель»

Если безоговорочно поверить рекомендации, представленной в этой главе, согласно которой отказ от использования методов чтения (getters) и записи (setters) свойств помогает создавать функциональные предметные модели, можно легко столкнуться с распространенной проблемой: проблемой извлечения данных из предметных моделей. Вам может понадобиться организовать отображение данных в пользовательском интерфейсе или пересылать их клиентам по электронной почте. Но как это реализовать, если необходимая информация будет скрыта внутри сущностей? Может показаться, что не остается ничего иного, как открыть доступ к методам чтения, и иногда это действительно лучший выбор. Однако шаблон «Хранитель» (memento) может предложить другой выход.

Шаблон «Хранитель» предполагает создание мгновенного снимка состояния сущности. Снимок — это черный ход в сущность, позволяющий использовать ее данные в других частях приложения, таких как пользовательский интерфейс. При этом у вас все еще сохраняются некоторые преимущества инкапсуляции, потому что структура состояния сущности остается закрытой. По сути, «хранители» предоставляют стабильный интерфейс для связи с другими частями приложения, сохраняя за вами возможность реорганизации внутренней структуры сущности.

Примером части предметной модели электронной коммерции, для которой был бы эффективен шаблон «Хранитель», может служить корзина покупателя, представленная в листинге 16.20.

Листинг 16.20. Поддержание инкапсуляции с помощью шаблона «Хранитель»

```
public class Basket
{
    // здесь нет общедоступных методов чтения/записи
    private int Id {get;set}
    private int Cost {get; set;}
    private List<Item> Items {get; set;}

    public void Add(Item item)
    {
        ...
    }

    public void Add(Coupon coupon)
    {
        ...
    }

    // внешний мир получает данные в виде BasketSnapshot
    public BasketSnapshot GetSnapshot()
    {
        return new BasketSnapshot(this.Id, this.Cost, this.Items.Count(),.....)
    }

    ...
}
```

Метод `Basket.GetSnapshot()` в листинге 16.20 возвращает снимок состояния сущности `Basket`, то есть «хранителя». Когда данные экспортируются таким способом, внутренние состояния сущности — `Id`, `Cost` и `Items` — остаются недоступными для остального приложения.

Отдавайте предпочтение функциям без побочных эффектов

Побочные эффекты усложняют осмысление программного кода и его тестирование и часто являются источником ошибок. В программировании вообще считается хорошим тоном избегать использования функций с побочными эффектами. В предыдущей главе вы могли даже видеть, что отсутствие побочных эффектов и неизменяемость являются двумя сильными сторонами объектов-значений. Но если уход от побочных эффектов считается хорошим советом, то исключение скрытых побочных эффектов является важнейшим правилом.

По всей вероятности, многие операции с сущностями должны производить некоторые побочные эффекты. В частности, многие методы сущности, вероятно, должны изменять ее внутреннее состояние. Но такой образ мышления может привести к вредной привычке наделять все методы сущностей побочными эффектами и, хуже того, скрытыми побочными эффектами. Следовательно, всегда нужно искать возможность заменить методы с побочными эффектами, такие как `Dice.Value()` в листинге 16.21, реализациями без побочных эффектов.

Листинг 16.21. Поведение сущности со скрытым побочным эффектом

```
public class Dice
{
    private Random r = new Random();

    public Dice(Guid id)
    {
        ...

        this.Id = id;
    }

    public Guid Id { get; private set; }

    // Плохо: выглядит как запрос, но каждый раз изменяется
    public int Value()
    {
        return r.Next(1, 7);
    }

    ...
}
```

Когда все внимание сконцентрировано на сокрытии внутреннего состояния и создании функциональных предметных моделей, легко написать код как в лис-

тинге 16.21, имеющий ненужные побочные эффекты. Клиенты сущности `Dice` наверняка предполагают, что метод `Value()` возвращает значение игрового кубика после последнего броска. Это объясняется тем, что имя `Value()` выглядит как запрос или свойство. Однако, как можно видеть, вызов `Value()` производит побочный эффект, изменяя значение. То есть клиент сущности `Dice` может дважды вызвать `Value()`, ожидая получить одно и то же значение, даже не подозревая, что он действует иначе.

В листинге 16.22 приводится более удачная реализация. В этом примере сущность `Dice` имеет метод `Roll()`. Имя `Roll()` выглядит как команда и предупреждает клиентов, что метод имеет побочные эффекты. Кроме того, тип `void` возвращаемого значения еще больше проясняет назначение метода. Обратите также внимание, что теперь `Value` — это свойство. Имя теперь не только выглядит как название операции чтения, но и действительно выполняет именно эту операцию, возвращая текущее значение кубика. Никаких скрытых побочных эффектов.

Листинг 16.22. Поведение сущности без скрытых побочных эффектов

```
public class Dice
{
    private Random r = new Random();

    public Dice(Guid id)
    {
        ...

        this.Id = id;
    }

    public Guid Id { get; private set; }

    // Хорошо: не изменяет внутреннего состояния
    public int Value { get; private set; }

    // Хорошо: выглядит как команда — побочный эффект ожидаем
    public void Roll()
    {
        Value = r.Next(1, 7);
    }

    ...
}
```

Ключевые идеи

- Сущности — это предметные понятия, имеющие уникальную индивидуальность в предметной области.
- Наличие жизненного цикла является еще одной отличительной чертой сущностей.

- Обсуждения со специалистами в предметной области — обычный способ выявления сущностей в предметной области, но при этом вы должны обращать внимание на используемые ими формулировки.
- Объекты-значения являются полной противоположностью сущностям; они не имеют индивидуальности и их равенство определяется значениями, которые они представляют.
- Принадлежность к сущностям и объектам-значениям определяется контекстом; сущности в одной предметной области могут оказаться объектами-значениями в другой.
- Выбор способа идентификации сущностей является важнейшей задачей реализации.
- Естественные ключи из предметной области, ключи, генерируемые приложением, и ключи, генерируемые хранилищем данных, — все это приемы идентификации сущностей.
- Сущности всегда должны находиться в состоянии, допустимом для данного контекста.
- Инварианты являются для сущностей непреложными истинами, поэтому они должны гарантироваться всегда.
- Функциональные возможности сущности должны соответствовать потребностям приложения. Моделирование поведения реального мира не является целью для сущностей.
- Будьте осторожны, моделируя физическое понятие в виде единственной сущности. Типичную сущность *Customer* часто можно логически разделить на множество сущностей в разных ограниченных контекстах.
- Старайтесь не использовать шаблон «Состояние» для моделирования жизненных циклов сущностей; часто этот путь ведет к сокрытию предметных понятий.
- Подумайте о возможности применения шаблонов, таких как «Хранитель», для поддержки функциональных предметных моделей, скрывающих организацию сущностей.

17

Службы предметной области

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в предметные службы
- Отличия от служб других типов
- Советы, когда предметным службам стоит отдать предпочтение
- Примеры предметных служб в разных предметных областях
- Обсуждение вопросов использования предметных служб внутри прикладного уровня и предметной модели с несколькими примерами и приемами

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 17 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Иногда при создании предметных моделей встречаются понятия или черты поведения, которые логически трудно выразить в виде сущностей или агрегатов. Это может служить неявным признаком необходимости создания службы предметной области (предметной службы).

К сожалению, термин *служба* (service) имеет слишком много смыслов. Тем не менее, с другой стороны, предметные службы легко отличить по двум определяющим характеристикам: они представляют предметные понятия и не имеют состояния. Обычно предметные службы используются для управления сущностями, а также в целях реализации бизнес-правил и не применяются для решения инфраструктурных задач; последнее оставляют на долю прикладных служб (application services).

Из этой главы вы узнаете, как создавать предметные службы для использования в разных бизнес-сценариях. В число этих сценариев входит экспортирование

предметных служб в качестве контрактов, реализованных за пределами предметной модели, и создание исключительно предметных служб, которые реализуют важные бизнес-правила и размещаются внутри предметной модели. В этой главе также будет показано несколько способов обращения к предметным службам из уровня служб и из предметной модели.

До того как продемонстрировать конкретные примеры, в этой главе будут определены основные предпосылки применения предметных служб и даны дополнительные разъяснения, что такие службы собой представляют.

Предметные службы

В этой книге содержится информация о сервис-ориентированной архитектуре (SOA), прикладных и предметных службах. Все эти понятия никак не связаны друг с другом, но многие разработчики все еще путаются в различиях между ними, о чем свидетельствуют обсуждения в списках рассылки. Концептуально предметные службы представляют предметные понятия; они реализуют черты поведения предметной области, выявляются в ходе общения со специалистами в предметной области и, конечно же, являются частью единого языка (UL). Если вы запомните эти важные характеристики, предметные службы станут для вас удобным и надежным инструментом.

Предметные службы имеют несколько важных технических характеристик, знание которых помогает реализовать их наиболее эффективно. Вы познакомитесь с ними чуть ниже.

Когда использовать предметные службы

Иногда в разговорах со специалистами обнаруживается некоторое предметное понятие, связанное с множеством сущностей, при этом нельзя с уверенностью сказать, какой из сущностей оно «принадлежит». Создается впечатление, что оно не принадлежит ни одной из них, и при попытке принудительно присвоить его той или иной сущности возникает ощущение неестественности. Это ощущение является явным признаком того, что вы имеете дело с предметной службой.

Инкапсуляция бизнес-правил и процессов

Основной задачей предметных служб является реализация некоторого поведения, основанного на сущностях или объектах-значениях. Это можно продемонстрировать с использованием двух сущностей *Competitor* и одной сущности *OnlineDeathmatch*, взятых из предметной модели игровой системы¹. Они представлены в листингах 17.1 и 17.2 соответственно.

¹ Здесь сущности *Competitor* представляют соперников, а сущность *OnlineDeathmatch* — бой, битву. — *Примеч. пер.*

Листинг 17.1. Сущность Competitor

```
public class Competitor
{
    ...

    public Guid Id {get; protected set;}

    public string GamerTag { get; protected set; }

    public Score Score { get; set; }

    public Ranking WorldRanking { get; set; }
}
```

Листинг 17.2. Сущность OnlineDeathmatch, содержащая логику подсчета очков

```
public class OnlineDeathmatch : IGame
{
    private Competitor player1;
    private Competitor player2;

    public OnlineDeathmatch(Competitor player1, Competitor player2, Guid id)
    {
        ...

        this.player1 = player1;
        this.player2 = player2;
        this.Id = id;
    }

    public Guid Id { get; protected set; }

    public void CommenceBattle()
    {
        ...

        // бой завершен

        // следующие инструкции в действительности
        // должны основываться на результатах игры
        var winner = player1;
        var loser = player2;

        UpdateScoresAndRewards(winner, loser);
    }

    private void UpdateScoresAndRewards(Competitor winner, Competitor loser)
    {
        // в настоящей игре также могут поддерживаться система рангов,
        // история боев, начисление премиальных очков, игровые действия,
        // сезонные акции, рекламные кампании
        winner.Score = winner.Score.Add(new Score(200));
        loser.Score = loser.Score.Subtract(new Score(200));
    }
}
```


По завершении боя каждый игрок получает некоторое число очков исходя из его действий в игре. Как один из вариантов, логику начисления очков и присвоения наград можно поместить в `OnlineDeathmatch`, как показано в листинге 17.2.

Одной существенной проблемой включения `UpdateScoresAndRewards()` в `OnlineDeathMatch` является большая гибкость правил; при определенных условиях владельцы игры могут пожелать удвоить присуждаемые очки или вручить специальный приз. В подобных случаях владельцы говорят, что функции начисления очков и присуждения наград сами по себе являются фундаментально важными понятиями. Точно такие же специальные акции часто проводятся в играх разных типов: `TeamBattle`, `CaptureTheFlag`, `GangCrusade` и т. д.

Этот сценарий указывает на необходимость создания предметной службы, отвечающей за реализацию правил начисления очков и присвоения наград. В листинге 17.4 показано определение интерфейсов `IGamingScorePolicy` и `IGamingRewardPolicy` предметной службы, которые можно внедрить и использовать в сущности `OnlineDeathmatch` или в любом другом объекте предметной области.

Листинг 17.3. Интерфейсы предметных служб

```
// интерфейсы предметной службы - часть UL
public interface IGamingScorePolicy
{
    void Apply(IGame game);
}

public interface IGamingRewardPolicy
{
    void Apply(IGame game);
}
```

Листинг 17.4. Использование предметных служб для управления сущностями

```
public class OnlineDeathmatch : IGame
{
    private Competitor player1;
    private Competitor player2;

    // предметные службы
    private IEnumerable<IGamingScorePolicy> scorers;
    private IEnumerable<IGamingRewardPolicy> rewarders;

    public OnlineDeathmatch(Competitor player1, Competitor player2,
        Guid id, IEnumerable<IGamingScorePolicy> scorers,
        IEnumerable<IGamingRewardPolicy> rewarders)
    {
        this.player1 = player1;
        this.player2 = player2;
        this.Id = id;
        this.scorers = scorers;
        this.rewarders = rewarders;
    }
}
```

```

public Guid Id { get; protected set; }

public void CommenceBattle()
{
    ...

    // бой завершен

    // следующие инструкции в действительности
    // должны основываться на результатах игры
    this.Winners.Add(player1);
    this.Losers.Add(player2);

    UpdateScoresAndRewards();
}

private void UpdateScoresAndRewards()
{
    foreach (var scorer in scorers)
    {
        scorer.Apply(this);
    }
    foreach (var rewarder in rewarders)
    {
        rewarder.Apply(this);
    }
}
}

```

Подход, использованный в листинге 17.4, позволяет переключать правила начисления очков и наград в соответствии с акциями, проводимыми организаторами. Другим важным моментом является то, что теперь можно сделать все эти понятия явными. Например, когда организаторы запустят акцию «годовая подписка за наибольшее число набранных очков», вы сможете смоделировать это явно, в виде предметной службы `Free12MonthSubscriptionForHighScoresRewardPolicy`, как показано в листинге 17.5.

Листинг 17.5. Конкретная реализация истинной предметной службы

```

// реализация предметной службы - части UL
public class Free12MonthSubscriptionForHighScoresReward : IGamingRewardPolicy
{
    private IScoreFinder repository;

    public Free12MonthSubscriptionForHighScoresReward(IScoreFinder repository)
    {
        this.repository = repository;
    }

    public void Apply(IGame game)
    {
        var top100Threshold = repository.FindTopScore(game, 100);
        if (game.Winners.Single().Score > top100Threshold)

```

```

    {
        // запустить процедуру награждения
    }
}

```

В листинге 17.5 показана предметная служба `Free12MonthSubscriptionForHighScoresRewardPolicy`, которая явно определяет важнейшее предметное понятие, существующее в едином языке (UL). Эта служба реализует бизнес-правила, существующие внутри предметной модели, и потому считается истинной (pure) предметной службой. Однако некоторые предметные службы реализуются за пределами предметной модели; их реализации находятся в уровне служб.

Представление контрактов

Другим типичным сценарием использования предметных служб является реализация контрактов — когда некоторое понятие имеет большое значение для предметной области, но его реализация опирается на инфраструктуру, которая не может использоваться в предметной модели. В качестве примера приведем службу `ShippingRouteFinder`, показанную в листинге 17.6. Она выполняет запрос к программному интерфейсу определения маршрутов, чтобы найти доступные маршруты к указанному пункту назначения.

Листинг 17.6. Реализация предметной службы, размещающаяся в уровне служб, а не в предметной модели

```
// реализации предметной службы может находиться в уровне служб
public class ShippingRouteFinder : IShippingRouteFinder
{
    public Route FindFastestRoute(Location departing, Location destination,
                                   DateTime departureDate)
    {
        // этот метод выполняет HTTP-вызов; поэтому данную службу
        // лучше определить за пределами предметной модели
        var response = QueryRoutingApi(departing, destination, departureDate);

        var route = ParseRoute(response);

        return route;
    }

    // ...
}

// интерфейс для предметной службы может быть определен в предметной модели
// это - "контракт"
public interface IShippingRouteFinder
{
    Route FindFastestRoute(Location departing, Location destination,
                           DateTime departureDate);
}
```

Обратите внимание, что логика, выполняющая HTTP-запросы к веб-серверу, не должна включаться в предметную область, потому что это — инфраструктурная задача; следовательно, реализация предметной службы `ShippingRouteFinder` из листинга 17.6 не может быть включена в предметную модель. Но данная задача является частью единого языка, поэтому интерфейс, определяющий ее, должен быть включен в модель.

Предметные службы можно использовать в качестве сценариев в самых разных случаях, включая следующие:

- идентификация сущностей;
- определение обменного курса;
- определение суммы налогов;
- рассылка извещений в режиме реального времени.

Внутреннее устройство предметной службы

Предметные службы обладают тремя основными техническими характеристиками: они представляют поведение и потому не имеют индивидуальности; они не имеют состояния; и они часто управляют множеством сущностей или объектов предметной области. В дополнение к примерам из предыдущего раздела в листинге 17.7 приводится реализация службы `RomanceOMeter`, подчеркивающая эти характеристики. Идеей для создания службы `RomanceOMeter` стало похожее понятие, существующее в предметной области сайтов знакомств, где оно используется для оценки совместимости потенциальных партнеров.

Листинг 17.7. Типичная предметная служба, которая не имеет состояния, представляет только поведение и управляет сущностями

```
// предметная служба - часть UI
public class RomanceOMeter
{
    // не имеет состояния - сопутствующие объекты, однако, допустимы

    // представляет только поведение
    public CompatibilityRating AssessCompatibility(LoveSeeker seeker1,
                                                LoveSeeker seeker2)
    {
        var rating = new CompatibilityRating(0);

        // управляет сущностями:
        // сравнивает истории свиданий, группу крови, образ жизни и др.
        if (seeker1.BloodType == seeker2.BloodType)
        {
            rating = rating + new CompatibilityRating(250);
        }

        ...

        // вернуть другой предметный объект (в данном случае объект-значение)
```

```
        return rating;
    }
}
```

Служба `RomanceOMeter` из листинга 17.7 не имеет никакого состояния, имеющего отношение к ее идентификации. Она представляет только поведение, реализующий единственный метод `AssessCompatibility()`, который вычисляет оценку совместимости. Как можно заметить, он не хранит никакого состояния и управляет несколькими сущностями `LoveSeeker`; в вычислениях участвуют только входные аргументы метода. Соответственно она удовлетворяет требованию об отсутствии состояния и демонстрирует, как предметные службы могут управлять другими объектами предметной области.

Избегайте шаблона «Анемичная предметная модель»

Выяснив, что не вся предметная логика должна находиться непосредственно в сущностях, а предметные службы являются весьма полезным понятием, будьте осторожны, чтобы не впасть в другую крайность и не включить слишком большой объем логики в предметные службы. Это может привести к созданию неточных, запутанных и плохо согласованных анемичных предметных моделей. Очевидно, что это может помешать получить некоторые основные выгоды DDD. Однако если соблюдать меру, это едва ли станет проблемой.

Поиск правильного баланса между слишком маленьким и слишком большим количеством предметных служб упрощается с накоплением опыта. Часто внимательность и логические рассуждения помогают принять правильное решение. Например, если при добавлении нового поведения в сущность возникает ощущение чужеродности, неправильности, плохой согласованности с тем, что говорят специалисты в предметной области, это может служить признаком необходимости реализовать его в виде предметной службы. Вы можете дополнительно обсудить проблему со специалистами и послушать, как они описывают данное понятие. Ссылаются ли они все время на сущность, говоря о понятии, или рассуждают о нем как о чем-то отдельном? С другой стороны, если для вас свойственно создавать слишком много предметных служб, возможно, вы слишком свободно мыслите.

Отличие от прикладных служб

Часто разработчики путают прикладные и предметные службы. Однако как только вы поймете концептуальные роли обеих, вы уже никогда не будете путаться. Как было показано в этой главе, предметные службы представляют понятия предметной области и, как минимум, их интерфейс определяется в предметной модели. Прикладные службы, напротив, не представляют предметных понятий и не содержат бизнес-правил. Также никакая из их частей не находится внутри предметной модели — ни реализации, ни интерфейс. Как будет показано в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования», прикладные службы находятся в уровне служб и решают инфраструктурные задачи, такие как транзакции, реализуя бизнес-сценарии использования.

На форумах DDD регулярно появляется вопрос: «Какой способ авторизации и аутентификации в предметной службе считается идеальным?». Ответ прост — никакой, потому что такие задачи должны решать прикладные службы. Это типичный пример путаницы, окружающей два типа служб, и знание роли каждого из них помогает понять, почему данный вопрос не имеет смысла (что, впрочем, понятно из-за неоднозначности термина *служба*).

Вы все еще можете полагать, что предметные и прикладные службы схожи между собой хотя бы тем, что и те и другие могут решать инфраструктурные задачи. Однако здесь также легко провести грань. Предметные службы опираются на инфраструктуру, которая используется для информирования предметной логики. Прикладные службы, решающие инфраструктурные задачи, напротив, лишь поддерживают нормальное функционирование предметной модели. Предметная служба может выполнять HTTP-запросы к веб-службам и записывать какие-то данные на диск, выполняя часть предметной логики, а прикладная служба оборачивает предметную модель транзакцией или создает соединения с базами данных, чтобы код мог действовать как единый сценарий использования.

ПРИМЕЧАНИЕ

Если у вас появится желание продолжить обсуждение различий между предметными и прикладными службами, посетите дискуссионный форум на сайте издательства Wrox <http://p2p.wrox.com/>.

Применение предметных служб

После знакомства с основами предметных служб и практическими примерами их реализации нам осталось понять, как ими пользоваться. Некоторые моменты требуют дополнительных пояснений, как, например, использование предметных служб внутри прикладных служб. Предметные службы часто используются как один из этапов процессов предметной области, целиком и полностью находящихся в предметной модели. Это вызывает споры, касающиеся внедрения предметных служб в сущности. Как вы увидите далее, это решенная проблема, но никакое решение не обходится без компромиссов и критики.

В уровне служб

Начнем с самого простого случая — использования предметных служб внутри прикладных служб. Обеспечивая поддержку сценария использования, прикладная служба может извлекать сущности из хранилища и передавать их предметной службе, как показано в листинге 17.8.

Листинг 17.8. Использование предметных служб и сущностей внутри прикладной службы

```
// прикладная служба
public class MultiMemberInsurancePremium
```

```
{
    private IPolicyRepository policyRepository;
    private IMemberRepository memberRepository;

    // предметная служба
    private IMultiMemberPremiumCalculator calculator;

    public MultiMemberInsurancePremium(IPolicyRepository policyRepository,
                                       IMemberRepository memberRepository,
                                       IMultiMemberPremiumCalculator calculator)
    {
        this.policyRepository = policyRepository;
        this.memberRepository = memberRepository;
        this.calculator = calculator;
    }

    public Quote GetQuote(int policyId, IEnumerable<int> memberIds)
    {
        var existingMainPolicy = policyRepository.Get(policyId);
        var additionalMembers = memberRepository.Get(memberIds);

        // передать сущности в предметную службу
        var multiMemberQuote = calculator.CalculatePremium(
            existingMainPolicy, additionalMembers
        );

        return multiMemberQuote;
    }
}
```

Прикладная служба `MultiMemberInsurancePremium` в листинге 17.8 объединяет `IMultiMemberPremiumCalculator` с сущностями `Policy` и `Member`, которые необходимо передать в вызов метода `CalculatePremium()` этой службы. Данный пример показывает, что на уровне служб предметные службы и объекты, такие как сущности, легко могут объединяться по мере необходимости. Однако такое удобство достижимо не всегда, как в случаях, когда сущности зависят от используемой предметной службы и объекты предметной области должны быть согласованы в предметной модели.

В предметной области

Иногда сущность нуждается в предметной службе, используя ее так, что это препятствует их объединению в прикладной службе. Типичным примером может служить отправка извещения после выполнения сущностью некоторой задачи. Этот сценарий демонстрирует листинг 17.9, где сущность `RestaurantBooking` генерирует событие `NotifyBookingConfirmation`, когда клиент подтверждает бронирование столика в ресторане. Сущность `RestaurantBooking` напрямую зависит от предметной службы `IRestaurantNotifier`, посылающей извещение о подтверждении бронирования.

Листинг 17.9. Сущность, зависящая от предметной службы

```
// сущность
public class RestaurantBooking
{
    ...

    public Guid Id { get; protected set; }

    public Restaurant Restaurant { get; protected set; }

    public Customer Customer { get; protected set; }

    public BookingDetails BookingDetails { get; protected set;}

    public bool Confirmed { get; protected set; }

    ...

    // поведение сущности, зависящее от предметной службы
    public void ConfirmBooking()
    {
        ...

        // restaurantNotifier – это предметная служба
        // можно ли получить ссылку на нее,
        // особенно если в работу вовлечен механизм ORM?
        restaurantNotifier.NotifyBookingConfirmation(
            Restaurant, Customer, Id, BookingDetails
        );

        ...
    }

    ...
}
```

Проблема реализации в листинге 17.9 заключается в получении ссылки на `restaurantNotifier`, экземпляра `IRestaurantNotifier`, внутри `ConfirmBooking()`. Может показаться, что ее легко решить простым внедрением зависимости через параметр конструктора. Однако все не так просто; часто для управления жизненным циклом сущностей используются механизмы объектно-реляционного отображения (ORM), лишаящие разработчиков возможности передавать зависимости через конструкторы объектов. Ниже перечислены приемы, каждый из которых решает эту проблему.

Связывание вручную

Если сущность или другой объект предметной области зависит от предметной службы, соответствующую ссылку на службу можно передать в конструктор, если создание объектов выполняется вручную. Листинг 17.10 демонстрирует решение, использующее фабричный метод для устранения этой проблемы.

Листинг 17.10. Передача ссылки на предметную службу через фабричный метод

```
public static class RestaurantBookingFactory
{
    // инициализировать объект - ORM здесь не используется,
    // поэтому нет нужды волноваться об этом
    public RestaurantBooking CreateBooking(Restaurant restaurant,
                                           Customer customer, BookingDetails details)
    {
        var id = Guid.NewGuid();
        var notifier = new RestaurantNotifier();
        return new RestaurantBooking(restaurant, customer, details, id, notifier);
    }
}
```

Листинг 17.10 демонстрирует, как передать предметную службу `RestaurantNotifier` в конструктор сущности `RestaurantBooking`, которая создается вручную в фабричном методе `CreateBooking()`. Шаблон «Фабрика» можно встретить во многих приложениях. На первый взгляд такое решение кажется обоснованным. Однако оно оказывается непригодным в DDD, когда сущности создаются автоматически, без вашего участия, например когда ORM загружает сущность из хранилища.

Конечно, можно предусмотреть обработку разных этапов жизненного цикла, которыми управляет механизм ORM, но подобные решения не всегда получаются красивыми. В листинге 17.11 показано, как фабричные методы могут добавлять второй этап конструирования сущностей, внедряющий ссылку на предметную службу в свойство сущности после ее создания механизмом ORM. Так же можно реализовать метод `Init()`, принимающий и внедряющий все зависимости, который будет действовать как псевдоконструктор. Однако в любом случае существует опасность забыть, что конструирование объектов должно выполняться в два этапа, из-за чего система может оказаться в противоречивом состоянии, а вы испытаете немало неприятных минут в поисках причин.

Листинг 17.11. Конструирование в два этапа, когда в работу вовлекается механизм ORM

```
public class RestaurantBookingRepository
{
    // интерфейс, предоставляемый механизмом ORM
    private ISession session;

    public RestaurantBookingRepository(ISession session)
    {
        this.session = session;
    }

    public RestaurantBooking Get(Guid restaurantBookingId)
    {
        // первый этап конструирования выполняется механизмом ORM
        var booking = session.Load<RestaurantBooking>(restaurantBookingId);

        // второй этап конструирования выполняется вручную
    }
}
```

```
        booking.Init(new RestaurantNotifier());  
        // как вариант: booking.Notifier = new RestaurantNotifier();  
    }  
}
```

Возможно, придется также позаботиться о других проблемах, связанных с конструированием в два этапа, например: не забыть последовательно применять выбранный шаблон во всем приложении и гарантировать, что `Init()` не будет вызван повторно после создания объекта.

Если настройка объектов вручную нежелательна, можно переложить часть обязанностей на механизм внедрения зависимостей, воспользовавшись контейнером, реализующим принцип инверсии управления (*Inversion of Control*, *IoC*).

Использование механизма внедрения зависимостей

Другой подход — внедрение предметных служб в сущности в виде зависимостей. После добавления предметной службы через параметр конструктора остается только связать желаемую реализацию с применением стандартного приема внедрения зависимостей. Использование внедрения зависимостей избавляет от необходимости вручную конструировать объекты. Тем не менее выбор данного или ручного подхода к связыванию зависимостей во многом лишь вопрос стиля, потому что проблема, связанная с использованием механизмов ORM, все еще может присутствовать.

Некоторые инструменты ORM поддерживают связывание зависимостей, управляемых используемым у вас контейнером *IoC*. Но, к сожалению, многие не имеют такой поддержки. Однако если вы все еще настроены использовать контейнер *IoC* и ORM, вам может пригодиться одно из потенциальных решений проблемы — шаблон проектирования «Локатор службы» (*Service Locator*).

Локатор службы

Прежде чем продолжить: шаблон «Локатор службы» является весьма спорным, и вскоре вы узнаете почему. Однако, как можно видеть в листинге 17.12, он решает поставленную задачу, позволяя контейнерам *IoC* настраивать зависимости сущностей, загружаемых механизмом ORM, без необходимости ручного вмешательства.

Листинг 17.12. Использование контейнеров *IoC* и механизмов ORM с применением шаблона «Локатор службы»

```
public class RestaurantBooking  
{  
    // жестко запрограммированное извлечение зависимости  
    // с использованием контейнера IoC container  
    private IRestaurantNotifier restaurantNotifier =  
        ServiceLocator.Get<IRestaurantNotifier>();  
  
    public Guid Id { get; protected set; }  
}
```

```
public Restaurant Restaurant { get; protected set; }

public Customer Customer { get; protected set; }

...

}
```

Жестко запрограммировав извлечение зависимости из класса `ServiceLocator`, как показано в листинге 17.12, можно загружать все зависимости сущности из контейнера `IoC` в процессе конструирования объекта после его загрузки механизмом `ORM`. Это устраняет необходимость выполнять конструирование вручную. Но, как известно, ничто не дается бесплатно; теперь сущность оказывается тесно связанной с классом `ServiceLocator` — инфраструктурным инструментом, который, в идеале, не должен загрязнять предметную модель своим присутствием.

Однако истинные проблемы, связанные с применением шаблона «Локатор служб», вытекают из образовавшейся тесной связи: например, теперь вам придется использовать в своих тестах фиктивные, или подставные, объекты (`mocks`). Другая проблема заключается в ухудшении очевидности зависимостей объекта, потому что они больше не передаются в конструктор.

Применение двойной диспетчеризации

Если бы вы были рады отказаться от передачи предметных служб в сущности на этапе конструирования, их можно передавать в аргументах методов, применив шаблон «Двойная диспетчеризация» (`Double Dispatch`). Согласно шаблону «Двойная диспетчеризация», предметная служба передается в вызов метода сущности, а сущность затем передает себя в вызов метода предметной службы, как показано в листинге 17.13.

Листинг 17.13. Использование двойной диспетчеризации для передачи предметных служб в сущности

```
public void ConfirmBooking(IRestaurantNotifier restaurantNotifier)
{
    ...

    Confirmed = restaurantNotifier.NotifyBookingConfirmation(this);

}
```

Как демонстрирует листинг 17.13, применение шаблона двойной диспетчеризации избавляет от необходимости передавать службы конструктору сущности. Вместо этого их можно передавать в вызовы методов сущности прикладной службой. Данный подход не пользуется большой популярностью по нескольким причинам. Некоторые считают, что передача зависимостей в методы накладывает на вызывающий код ответственность, которую он не должен нести. Другие утверждают, что зависимости обычно передаются через конструктор и не долж-

ны быть частью сигнатур методов; они отвечают за реализацию деталей, которые могут измениться, тогда как сигнатуры методов не должны быть настолько изменчивыми.

По этим причинам двойная диспетчеризация менее популярна среди разработчиков, хотя и не настолько, как другие варианты, представленные выше. В любом случае вам стоит исследовать идею самостоятельно и оценить, насколько хорошо она подходит для вашего проекта. Впрочем, существует еще одно решение, ставшее популярным, — использование шаблона «Предметные события» (Domain Events), позволяющего по-настоящему отвязать сущности от предметных служб.

Ослабление связей с помощью предметных событий

Шаблон «Предметные события» представляет большой интерес, потому что полностью исключает необходимость внедрения предметных служб в сущности. Когда сущность выполняет какие-то важные действия, она может послать предметное событие, которое будет обработано подписчиками, зарегистрировавшимися для получения этого события. Как вы уже наверняка догадались, в этом случае предметные службы могут располагаться внутри подписчиков, а не в сущностях.

В листинге 17.14 демонстрируется сущность `RestaurantBooking`, посылающая предметное событие `BookingConfirmedByCustomer`. В листинге 17.15 представлен обработчик события, или подписчик, обрабатывающий события этого типа, вызывая предметную службу `IRestaurantNotifier`. Наконец, в листинге 17.16 показано, как мало требуется, чтобы связать эти два компонента.

Листинг 17.14. Сущность, посылающая предметное событие

```
public class RestaurantBooking
{
    public Guid Id { get; protected set; }

    ...

    public void ConfirmBooking()
    {
        ...

        DomainEvents.Raise(new BookingConfirmedByCustomer(this));

        ...
    }

    ...
}
```

Листинг 17.15. Предметная служба, используемая обработчиком событий

```
// domain event handler
public class NotifyRestaurantOnCustomerBookingConfirmation :
    IHandleEvents<BookingConfirmedByCustomer>
```

```
{
    private IRestaurantNotifier restaurantNotifier;

    public NotifyRestaurantOnCustomerBookingConfirmation(
        IRestaurantNotifier restaurantNotifier)
    {
        this.restaurantNotifier = restaurantNotifier;
    }

    public void Handle(BookingConfirmedByCustomer @event)
    {
        // теперь ресторан не управляет своим состоянием в полной мере,
        // зато сущности не требуется знать о существовании предметной службы
        var booking = @event.RestaurantBooking;
        booking.Confirmed =
            restaurantNotifier.NotifyBookingConfirmation(
                booking.Restaurant, booking.Customer,
                booking.Id, booking.BookingDetails
            );
    }
}
```

Листинг 17.16. Подписка обработчика для получения предметного события

```
var notifier = new RestaurantNotifier();
var handler = new NotifyRestaurantOnCustomerBookingConfirmation(notifier);
DomainEvents.Regisiter<BookingConfirmedByCustomer>(handler);
```

В листингах с 17.14 по 17.16 представлен минимально возможный пример реализации шаблона «Предметное событие» для сценария бронирования столика в ресторане. Этому шаблону полностью посвящена глава 18 «События предметной области», где представлены более полные примеры, демонстрирующие сильные и слабые стороны шаблона. Один из очевидных недостатков, который вы, возможно, подметили в листингах с 17.14 по 17.16, заключается в распределении логики между сущностью `RestaurantBooking` и обработчиком `NotifyRestaurantOnCustomerBookingConfirmation`, тогда как в других шаблонах ее можно было реализовать в виде одного фрагмента последовательно выполняющегося кода.

Должны ли сущности знать о существовании предметных служб?

В предыдущем примере можно видеть, как предметные события избавляют от необходимости внедрять предметные службы в сущности, тогда как все остальные шаблоны предлагают совершенно противоположное решение и стремятся найти способ сделать такое внедрение возможным. Последнее считается весьма спорным подходом в сообществе DDD; многие практики утверждают, что это неудачная идея. Как бы то ни было, принимая решение о выборе того или иного подхода, вы должны исходить из конкретных условий, личных предпочтений и опыта.

Ключевые идеи

- Иногда поведение не принадлежит какой-то сущности или объекту-значению, но все еще представляет важное предметное понятие; это может служить подсказкой о необходимости реализовать предметную службу.
- Предметные службы представляют предметные понятия; они являются частью единого языка.
- Предметные службы часто используются для управления сущностями и объектами-значениями в ходе выполнения операций, не имеющих состояния.
- Слишком большое число предметных служб может привести к созданию анемичной предметной модели, плохо согласующейся с предметной областью.
- Слишком малое число предметных служб может привести к ошибочному включению логики в сущности или объекты-значения. Это может вызвать смешение разных понятий и ухудшить ясность модели.
- Предметные службы также используются как контракты в случаях, когда интерфейс определяется в предметной модели, а реализация — нет.
- Когда сущность зависит от предметной службы, на выбор есть множество вариантов реализации такой зависимости, включая внедрение зависимостей, двойную диспетчеризацию и предметные события.

18

События предметной области

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в шаблон «Предметные события»
- Знакомство с основными понятиями, связанными с предметными событиями
- Множество шаблонов реализации предметных событий
- Тестирование приложений, использующих предметные события

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.worox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 18 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Практики DDD пришли к выводу, что способны лучше понять предметную область, исследуя не только ее сущности, но и события, которые в ней возникают. Эти события, которые называют *событиями предметной области*, или *предметными событиями* (domain events), выявляются в ходе сеансов переработки знаний со специалистами в предметной области. Выявление предметных событий настолько ценно, что практики DDD ввели новые приемы переработки знаний для лучшей фокусировки на событиях, добавив такие методики, как событийный шторм (<http://ziobrando.blogspot.co.uk/2013/11/introducing-event-storming.html#.U5YPynU-mHs>). Однако новые методики влекут за собой новые проблемы. Теперь, когда концептуальные представления стали событийно-ориентированными, программный код также должен быть событийно-ориентированным, чтобы описать концептуальную модель. Именно здесь особую ценность приобретает шаблон проектирования «Предметные события».

Во второй части книги рассказывалось о передаче событий в виде асинхронных сообщений между ограниченными контекстами с использованием таких транспортных, как шина сообщений или архитектура REST. Однако в общем случае шаблон проектирования «Предметные события» используется как шаблон для однопоточной среды выполнения внутри предметной модели, в рамках одного

ограниченного контекста. В этой главе вы познакомитесь с примерами, демонстрирующими разные способы реализации предметных событий и типичные компромиссы, на которые приходится идти при их использовании.

Прежде чем двинуться дальше, важно отметить, что использование предметных событий не требует применения приема регистрации событий (event sourcing). К сожалению, это распространенное заблуждение; в целом вполне возможно использовать традиционные механизмы хранения данных, такие как базы данных SQL. Однако применение регистрации событий или асинхронных сообщений в сочетании с предметной моделью, использующей предметные события, часто оказывается неплохой комбинацией. Подробнее об этой связи рассказывается в главе 22 «Регистрация событий».

На протяжении всей этой главы каждый шаблон реализации предметных событий будет использоваться для воплощения одного и того же сценария из рабочего процесса службы доставки пиццы. Это поможет вам проводить прямые параллели. Сценарий, о котором пойдет речь, — это моделирование процесса гарантированной доставки, когда клиент получает скидку, если пицца не была доставлена за определенное время.

ВНИМАНИЕ

Предметные события возникают в предметной области, а шаблон «Предметные события» описывает способ их моделирования с помощью программного обеспечения. То есть теоретически термин «предметные события» может применяться к обоим этим понятиям. Эта неоднозначность иногда может приводить к путанице. В данной главе под этим термином будет подразумеваться шаблон проектирования, если явно не указано иное.

Суть шаблона «Предметные события»

Если вы понимаете суть шаблона «Публикация/подписка» или событий C#, вы быстро освоите шаблон «Предметные события». Суть этого шаблона заключается в использовании инфраструктурного компонента, иногда внутри предметной модели, для публикации событий. По умолчанию события затем синхронно обрабатываются в том же потоке выполнения каждым обработчиком, зарегистрированным для получения событий данного типа. Этот шаблон может также применяться для обработки асинхронных событий.

Важные предметные происшествия, которые уже случились

События — это всего лишь неизменяемые классы с общедоступными свойствами (объекты данных, «простые старые объекты C#» (Plain Old C# Objects, POCO), «простые старые объекты Java» (Plain Old Java Objects, POJO)), представляющие важные события, происходящие в предметной области. В листинге 18.1 представлено событие `DeliveryGuaranteeFailed`, которое возникает в предметной области, когда пицца доставляется после истечения установленного времени.

Листинг 18.1. Событие

```
public class DeliveryGuaranteeFailed
{
    public DeliveryGuaranteeFailed(OrderForDelivery order)
    {
        Order = order;
    }

    public OrderForDelivery Order { get; private set; }
}
```

Обратите внимание, что событие `DeliveryGuaranteeFailed` в листинге 18.1 — это простой класс со свойствами; это все, что необходимо для представления события.

Реакция на события

В ответ на события выполняются обработчики событий. В листинге 18.2 показано, как можно зарегистрировать обработчик событий, чтобы обеспечить вызов метода (`onDeliveryFailure`), когда возникнет событие (`DeliveryGuaranteeFailed`).

Листинг 18.2. Обработчик событий

```
public void Confirm(DateTime timeThatPizzawasDelivered, Guid orderId)
{
    using(DomainEvents.Register<DeliveryGuaranteeFailed>(onDeliveryFailure))
    {
        var order = orderRepository.FindBy(orderId);
        order.ConfirmReceipt(timeThatPizzawasDelivered);
    }
}

private void onDeliveryFailure(DeliveryGuaranteeFailed evt)
{
    // здесь выполняется обработка предметного события
}
```

Возможная асинхронность

Шаблон «Предметные события» вполне можно использовать с процессами, протекающими асинхронно, включая обмен сообщениями между ограниченными контекстами. Как было показано во второй части книги, посвященной вопросам интеграции ограниченных контекстов, обмен асинхронными сообщениями между ограниченными контекстами является современным решением, помогающим повысить надежность и масштабируемость. Вы можете запускать эти процессы из обработчиков предметных событий. Иногда даже бывает желательно использовать асинхронные надежные взаимодействия внутри ограниченного контекста, например, для реализации потенциально непротиворечивых агрегатов. Слабосвязанная природа предметных событий может помочь в обоих сценариях.

Важно отметить, что при создании обработчиков событий, запускающих асинхронные процессы, необходимо четко соблюдать границы транзакций. Например, если один обработчик событий вносит изменения в базу данных, а другой публикует сообщение, может потребоваться отменить обе операции, если одна из них потерпит неудачу, как показано на рис. 18.1.

ПРИМЕЧАНИЕ

Агрегаты — это группы связанных друг с другом сущностей и объектов-значений. Подробнее об агрегатах рассказывается в главе 19 «Агрегаты».

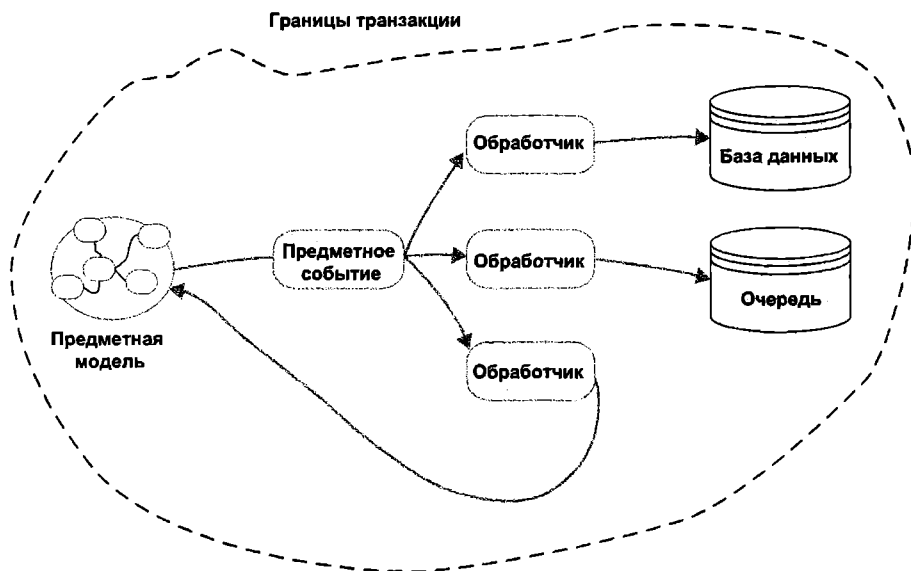


Рис. 18.1. Обеспечение правильного поведения транзакций

Внутренние и внешние события

При использовании шаблона «Предметные события» необходимо помнить об одном важном различии, чтобы избежать путаницы, которая может привести к неудачной технической реализации. Крайне важно различать внутренние и внешние события. Внутренние события происходят внутри предметной модели — они не передаются между ограниченными контекстами. В этой главе вы узнаете, как шаблон «Предметные события» использует внутренние события. С внешними событиями вы уже познакомились во второй части книги.

Разграничивать внутренние и внешние события важно потому, что они обладают разными характеристиками. Внутренние события действуют в узкой области, определяемой рамками одного ограниченного контекста, поэтому они могут быть связаны с предметными объектами, как это показано в листинге 18.1. Это не представляет никакой угрозы, потому что не влечет образования тесной связи между

предметными объектами и другими ограниченными контекстами. Внешние события, напротив, часто имеют плоскую организацию, экспортируя лишь несколько свойств, в основном связующие идентификаторы, как показано в листинге 18.3.

Листинг 18.3. Внешние события не могут представлять предметные объекты

```
public class DeliveryGuaranteeFailed
{
    public DeliveryGuaranteeFailed(Guid orderId)
    {
        this.OrderId = orderId;
    }

    public Guid OrderId { get; private set; }
}
```

Во второй части книги вы узнали, что внешним событиям необходимо присваивать версии во избежание нарушения работоспособности программы из-за изменений. Это еще одно отличие внешних событий от внутренних — если внести такие разрушительные изменения во внутреннее событие, программный код просто перестанет компилироваться (если используется компилируемый язык программирования). То есть для внутренних событий не требуется поддерживать механизм версий.

Приступив к реализации предметных событий, вы увидите, что в типичных сценариях использования может возникать множество внутренних событий и только одно или два внешних, вызванных в уровне служб. Рисунок 18.2 иллюстрирует последовательность событий, возникающих в типичном сценарии использования.

Учитывая все эти различия, имеет смысл поместить события в разные пространства имен, чтобы провести грань между внутренними и внешними событиями.

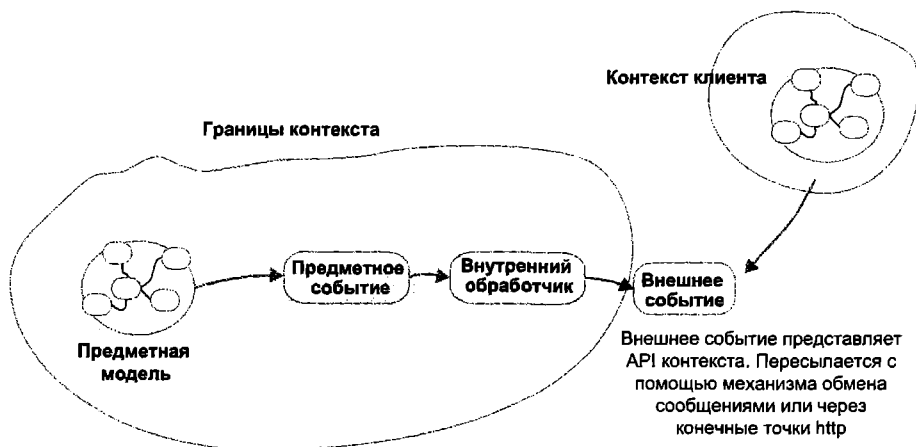


Рис. 18.2. Поток внутренних и внешних событий, возникающих в типичном сценарии использования

Обработка событий

Прежде чем перейти к конкретным примерам реализации шаблона «Предметные события», важно получить представление о различиях между обработчиками в уровнях предметных и прикладных служб. Даже при том, что они выглядят одинаково, области их ответственности существенно отличаются. Обработчики предметных событий вызывают предметную логику, например предметные службы, тогда как обработчики событий в уровне прикладных служб по своей природе ближе к решению инфраструктурных задач, таких как отправка электронной почты или публикация событий, предназначенных для других ограниченных контекстов.

Вызов предметной логики

Обработчики событий, находящиеся внутри предметной модели, могут обрабатывать события, возникающие там. К таким сценариям относится моделирование последовательностей взаимодействий, происходящих в предметной области. Например, на сайте сервиса доставки готовой еды заказ сначала проверяется на выполнимость, затем проверяется, не включен ли клиент в черный список из-за отказа оплатить предыдущие заказы. Далее механизм проверки генерирует собственные события, чтобы запустить следующий этап бизнес-процесса. Очень часто можно увидеть, что обработчики предметных событий делегируют выполнение основной работы предметным службам.

Вызов прикладной логики

Присутствие обработчиков событий в уровне прикладных служб, в дополнение к обработчикам в предметной модели, дает определенные выгоды. Эти обработчики обычно призваны решать инфраструктурные задачи, такие как отправка электронной почты. Обратите внимание, что эти обработчики не являются частью единого языка или предметной области.

Одной из важнейших задач, которые решают обработчики в уровне прикладных служб, является поддержка взаимодействий с внешними ограниченными контекстами посредством технологий, представленных во второй части книги. Они также отвечают за взаимодействия с внешними службами, такими как системы приема платежей.

Варианты реализации шаблона «Предметные события»

Предметные события — это обобщенный шаблон, который просто добавляет предметную семантику в шаблон «Издатель/подписчик». Это дает большую свободу выбора при его реализации. В следующих примерах вы познакомитесь с самыми разными решениями — от синхронных, использующих встроенные конструкции языка, до альтернатив на основе шины сообщений.

Используйте модель событий .Net Framework

Самый простой способ реализации шаблона «Предметные события» — использовать встроенные средства, предоставляемые языком или платформой, с помощью которых вы ведете разработку. В языке C#, например, существует ключевое слово `event`.

Чтобы определить предметное событие с помощью ключевого слова `event`, нужно создать в классе сущности общедоступное поле с именем, соответствующим имени предметного события. Также нужно создать делегата, представляющего контракт события. Обе эти детали показаны в листинге 18.4.

Листинг 18.4. Определение предметного события с ключевым словом `event` в языке C#

```
public class OrderForDelivery
{
    ...

    public delegate void
        DeliveryGuaranteeFailedHandler(DeliveryGuaranteeFailed evtnt);

    public event DeliveryGuaranteeFailedHandler DeliveryGuaranteeFailed;

    ...
}
```

В листинге 18.4 с помощью ключевого слова `event` создается событие с именем `DeliveryGuaranteeFailed`. Обратите внимание, что это событие имеет тип `DeliveryGuaranteeFailedHandler`, указывающий, что все обработчики события `DeliveryGauranteeFailed` должны иметь такую же сигнатуру, как `DeliveryGuaranteeFailedHandler`. Сигнатуру `DeliveryGuaranteeFailedHandler` можно увидеть строкой выше, где она объявляется с помощью ключевого слова `delegate`. Эта сигнатура является функцией, не возвращающей результат, с единственным параметром типа `DeliveryGuaranteeFailed`. Обработчик с этой сигнатурой и его регистрация демонстрируются в листинге 18.5.

Листинг 18.5. Подписка на встроенные события C#

```
public class ConfirmDeliveryOfOrder
{
    private IOrderRepository orderRepository;
    private IBus bus;

    public ConfirmDeliveryOfOrder(IOrderRepository orderRepository, IBus bus)
    {
        this.orderRepository = orderRepository;
        this.bus = bus;
    }

    public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
    {
```

```

        var order = orderRepository.FindBy(orderId);
        order.DeliveryGuaranteeFailed += onDeliveryGuaranteeFailed;
        order.ConfirmReceipt(timeThatPizzaWasDelivered);
    }

    private void onDeliveryGuaranteeFailed(DeliveryGuaranteeFailed evnt)
    {
        bus.Send(new RefundDueToLateDelivery() { OrderId = evnt.Order.Id });
    }
}

```

`onDeliveryGuaranteeFailed` — это функция, не возвращающая результат и имеющая единственный параметр типа `DeliveryGuaranteeFailed`. Следовательно, его можно зарегистрировать в роли обработчика события `DeliveryGuaranteeFailed` как показано в листинге 18.5. Регистрация выполняется с применением специального синтаксиса C#: `+=`.

Чтобы послать событие C#, его нужно вызвать как метод, как показано в листинге 18.6, где генерируется событие `DeliveryGuaranteeFailed`. Важно сначала убедиться, что поле, соответствующее событию, не является пустой ссылкой, потому что в отсутствие подписчиков на событие это поле будет содержать `null` и попытка использовать его вызовет исключение обращения по пустой ссылке.

Листинг 18.6. Вызов встроенного события C#

```

public class OrderForDelivery
{
    ...

    public delegate void
        DeliveryGuaranteeFailedHandler(DeliveryGuaranteeFailed evnt);

    public event DeliveryGuaranteeFailedHandler DeliveryGuaranteeFailed;

    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                if (DeliveryGuaranteeFailed != null)
                    DeliveryGuaranteeFailed(new DeliveryGuaranteeFailed(this));
            }
        }
    }
}

```

Чтобы запустить асинхронную обработку, можно создать и зарегистрировать обработчик событий, публикующий асинхронные сообщения и записывающий их в очередь или в базу данных. При этом вы должны убедиться в выполнении всех операций в границах текущей транзакции, чтобы в случае отмены транзакции были отменены все изменения, выполненные в ходе асинхронной обработки. Те, кто использует C# и ключевое слово `event`, наверняка захотят настроить координатор распределенных транзакций (Distributed Transaction Coordinator, DTC).

ВНИМАНИЕ

Обработчики событий, решающие инфраструктурные задачи, такие как операции с базами данных, должны регистрироваться в уровне прикладных служб, чтобы предотвратить проникновение технических деталей в предметную модель. Как вариант, в предметной модели можно определить контракт предметной службы (за подробностями обращайтесь к главе 17 «Службы предметной области»).

Как можно видеть в листингах с 18.4 по 18.6, для использования встроенной модели событий на основе ключевого слова `event` не требуется тратить много времени на предварительное изучение. Однако этот подход имеет свою цену; между издателем и подписчиками образуется тесная связь. Это можно наблюдать в листинге 18.5, где видно, что обработчик события должен знать, какой объект публикует события. В следующем разделе представлена альтернативная реализация шаблона «Предметные события» со слабой связью между издателями и подписчиками.

Использование шины памяти

Чтобы избежать образования тесной связи между издателями и подписчиками, шаблон «Предметные события» можно реализовать с использованием шины сообщений в памяти. Это даст возможность организовать асинхронное выполнение некоторых операций, если это будет иметь смысл и шина сообщений поддерживает синхронную и асинхронную транспортировку. К числу таких шин, поддерживающих оба варианта, относится `NServiceBus`.

ПРИМЕЧАНИЕ

Примеры в этом разделе написаны для версии `NServiceBus 4.3.3`, которая доступна для загрузки по адресу <https://github.com/Particular/NServiceBus/releases/download/4.3.3/Particular.NServiceBus-4.3.3.exe>. Вам также потребуется добавить ссылку на `NServiceBus` в свой проект, для чего следует выполнить следующую команду в консоли диспетчера пакетов `NuGet` (не забудьте указать имя своего проекта):

```
Install-Package NServiceBus.Host -ProjectName YourProjectNameHere -Version 4.3.3
```

При использовании шины сообщений вновь в центре внимания оказываются события и обработчики. Но теперь, вместо связывания их друг с другом, вам придется публиковать события и регистрировать обработчики в шине. Однако в действительности при использовании `NServiceBus` нет необходимости регистри-

ровать подписчиков, потому что все обработчики событий реализуют интерфейс `IHandleMessages<T>`, что определен в `NServiceBus`, где `T` — это тип обрабатываемого события. Таким образом, `NServiceBus` может автоматически обнаруживать обработчики событий. Достаточно просто сообщить `NServiceBus`, какие классы являются событиями, как показано в листинге 18.7.

ПРИМЕЧАНИЕ

Рекомендуется выполнять настройку `NServiceBus` в уровне служб, чтобы избежать технических деталей в предметной модели. Пока проект с настройками ссылается на проект предметной модели, соглашения будут обнаруживать события.

Листинг 18.7. Настройка автоматической регистрации подписчиков в `NServiceBus`

```
public class EndpointConfig : IConfigureThisEndpoint,
    AsA_Server, AsA_Publisher, IWantCustomInitialization
{
    public void Init()
    {
        Configure.With()
            .DefiningEventsAs(t => t.Namespace != null
                && t.Namespace.Contains("Events"));
    }
}
```

Программный код в листинге 18.7 использует соглашение, в соответствии с которым любой класс в пространстве имен с названием `Events` будет рассматриваться фреймворком `NServiceBus` как событие. Таким образом, если опубликовать событие, представленное в листинге 18.8 (обратите внимание, что пространство имен содержит `Events`), `NServiceBus` вызовет обработчик, что показан в листинге 18.9.

Листинг 18.8. Событие `NServiceBus`

```
namespace OnlineTakeawayStore.NServiceBus.Model.Events
{
    public class DeliveryGuaranteeFailed
    {
        public DeliveryGuaranteeFailed(OrderForDelivery order)
        {
            Order = order;
        }

        public OrderForDelivery Order { get; private set; }
    }
}
```

Листинг 18.9. Обработчик событий `NServiceBus`

```
// Автоматически обнаруживается фреймворком NServiceBus благодаря
// наследованию IHandleMessages<T>
public class RefundOnDeliveryGuaranteeFailureHandler :
    IHandleMessages<DeliveryGuaranteeFailed>
{
    // ...
}
```



```
// Внедряется фреймворком NServiceBus
public IBus Bus { get; set; }

public void Handle(DeliveryGuaranteeFailed message)
{
    Bus.Send(new RefundDueToLateDelivery() { OrderId = message.Order.Id });
}
}
```

Далее, в листинге 18.10, показано, как это сообщение публикуется в шине NServiceBus. Отметьте, что подписчик не знает и не заботится о том, кто является издателем события, и наоборот.

Листинг 18.10. Публикация событий в шине NServiceBus

```
public class OrderForDelivery
{
    ...

    private IBus Bus { get; set; }

    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                Bus.InMemory.Raise(new DeliveryGuaranteeFailed(this));
            }
        }
    }

    ...
}
```

Одна из сложностей, связанных с использованием шины сообщений, — передача ее в предметную модель. В идеале нежелательно, чтобы такие технические детали, как шины сообщений, проникали в предметную модель. С другой стороны, должен быть какой-то способ публикации событий. Для некоторых разработчиков это веский повод пойти на компромисс и позволить некоторым техническим деталям проникнуть в предметную область, потому что это позволит увеличить выразительность предметных понятий в модели.

По умолчанию NServiceBus рассылает сообщения в асинхронном режиме, поэтому если вы решили, что некоторые предметные события должны передаваться асинхронно, вы относительно легко сможете выполнить переход. Для этого достаточно добавить новый обработчик события, подписанный на события в памяти

и публикующий асинхронные события. Также можно сразу публиковать события асинхронно, используя `Bus.Publish()` вместо `Bus.InMemory.Raise()`, благодаря чему все подписчики будут вызваны асинхронно в отдельном потоке выполнения.

Принимая решение о публикации асинхронных событий, важно изучить транзакционные требования. Если два обработчика событий должны действовать атомарно — либо преуспеть вместе, либо вместе потерпеть неудачу, — вы должны гарантировать синхронное их выполнение. В противном случае высока вероятность столкнуться с ошибками, вызванными противоречивым состоянием.

ПРИМЕЧАНИЕ

Если для вас нежелательно, чтобы предметная модель оказалась привязана к какой-то определенной технологии обмена сообщениями, можно инкапсулировать ее за интерфейсами. Далее вы увидите статический класс `DomainEvents`, который можно рассматривать как один из вариантов.

Статический класс `DomainEvents` Уди Дахана

Традиционно философия DDD рекомендует исключать любые инфраструктурные задачи из предметной модели. Классические руководства по объектно-ориентированному программированию предупреждают о необходимости избегать полагаться на статические методы, потому что они способствуют образованию тесных связей. В связи с этим вы, возможно, удивитесь, услышав, что еще одна популярная версия шаблона «Предметные события» опирается на статический класс `DomainEvents`, который решает инфраструктурные задачи и находится в предметной модели. В листинге 18.11 демонстрируется реализация статического класса `DomainEvents`, основанного на работе эксперта в DDD Уди Дахана (<http://www.udidahan.com/2008/08/25/domain-events-take-2/>).

Листинг 18.11. Статический класс `DomainEvents`

```
public static class DomainEvents
{
    [ThreadStatic]
    private static List<Delegate> _actions;
    private static List<Delegate> Actions
    {
        get
        {
            if (_actions == null)
            {
                _actions = new List<Delegate>();
            }
            return _actions;
        }
    }

    public static IDisposable Register<T>(Action<T> callback)
    {

```

```

        Actions.Add(callback);
        return new DomainEventRegistrationRemover(
            () => Actions.Remove(callback)
        );
    }

    public static void Raise<T>(T eventArgs)
    {
        foreach (Delegate action in Actions)
        {
            Action<T> typedAction = action as Action<T>;
            if (typedAction != null)
            {
                typedAction(eventArgs);
            }
        }
    }

    private sealed class DomainEventRegistrationRemover : IDisposable
    {
        private readonly Action _callOnDispose;
        public DomainEventRegistrationRemover(Action toCall)
        {
            _callOnDispose = toCall;
        }

        public void Dispose()
        {
            _callOnDispose();
        }
    }
}

```

Класс `DomainEvents` имеет два важных метода: `Register<T>` регистрирует функцию обратного вызова, которая должна быть выполнена, когда событие типа `T` будет возбуждено другим важным методом: `Raise<T>`.

Еще одной особенностью в листинге 18.11, на которую стоит обратить внимание, является использование приватного класса `DomainEventsRegistrationRemover`. Метод `Raise<T>` использует экземпляр этого класса, чтобы автоматически отменять регистрацию обработчиков событий после первого же их выполнения.

В листинге 18.12 показано, как зарегистрировать обработчик события с помощью `DomainEvents`, а в листинге 18.13 затем показано, как публиковать события.

Листинг 18.12. Регистрация обработчика событий с помощью `DomainEvents`

```

public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{
    using(DomainEvents.Register<DeliveryGuaranteeFailed>(onDeliveryFailure))
    {
        var order = orderRepository.FindBy(orderId);
        order.ConfirmReceipt(timeThatPizzaWasDelivered);
    }
}

```

```

}

private void onDeliveryFailure(DeliveryGuaranteeFailed evnt)
{
    // обрабатывает внутреннее и публикует внешнее событие
    bus.Send(new RefundDueToLateDelivery() { OrderId = evnt.Order.Id });
}

```

Листинг 18.13. Публикация события с помощью DomainEvents

```

public class OrderForDelivery
{
    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered)
            )
            {
                DomainEvents.Raise(new DeliveryGuaranteeFailed(this));
            }
        }
    }
}

```

Решение проблем многопоточного выполнения

Одной из больших сложностей, связанных с использованием статического класса `DomainEvents`, является проблема многопоточного выполнения. Это обусловлено тем, что обработчики регистрируются в одном потоке, а вызываются, когда события обрабатываемого ими типа публикуются, — в другом. Это следствие того, что `DomainEvents` является статическим классом; все потоки выполнения используют один и тот же экземпляр.

Избежать этой проблемы помогает применение атрибута `ThreadStatic` к коллекциям обработчиков и функций обратного вызова в классе `DomainEvents`. При использовании этого атрибута каждый поток выполнения получит собственную коллекцию, то есть обработчики, зарегистрированные в одном потоке выполнения, не будут доступны в другом. В листинге 18.14 показано, как использовать атрибут `ThreadStatic` с предметными событиями.

Листинг 18.14. Применение атрибута ThreadStatic к коллекции обработчиков и функций обратного вызова в DomainEvents

```

public static class DomainEvents
{
    [ThreadStatic] // каждый поток получит свою коллекцию

```

```
private static List<Delegate> _actions;

...

}
```

Однако такое решение вызывает проблемы в веб-приложениях ASP.NET. Скотт Хансельман (Scott Hanselman) разъясняет в своем блоге (<http://www.hanselman.com/blog/AtaleOfTwoTechniquesTheThreadStaticAttributeAndSystemWebHttpContextCurrentItems.aspx>), почему атрибут `ThreadStatic` никогда не должен использоваться в приложениях ASP.NET.

Замена статического класса внедрением в методы

Если вы не готовы связать свою предметную модель со статическим классом для публикации событий, против чего, кстати, возражают многие практики DDD, вы можете пойти другим путем и передавать подобные классы в предметную модель через аргументы методов. Данный подход также стоит рассмотреть, когда желательно избежать проблем с атрибутом `ThreadStatic`, описанных выше. В листинге 18.15 показано, как передать экземпляр диспетчера событий в предметный метод и использовать его.

Листинг 18.15. Использование нестатического класса `DomainEvents` с приемом внедрения в методы

```
public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered,
    IEventDispatcher dispatcher)
{
    if (Status != FoodDeliveryOrderSteps.Delivered)
    {
        TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
        Status = FoodDeliveryOrderSteps.Delivered;
        if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
            TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered)
        )
        {
            dispatcher.Raise(new DeliveryGuaranteeFailed(this));
        }
    }
}
```

Возврат предметных событий

Другой задачей шаблона «Предметные события» является разделение публикации и обработки событий, чтобы изолировать возможные побочные эффекты. Этот подход реализуется сохранением коллекции событий в корне агрегата и их публикацией после того, как корень агрегата завершит свою операцию. Примечательно, что в данном случае для публикации событий вызывается диспетчер из уровня служб, что предотвращает проникновение технических деталей в предметную модель.

В листинге 18.16 показана альтернативная версия сущности `OrderForDelivery`, хранящей коллекцию событий (свойство `RecordedEvents`). Каждый раз, когда сущность загружается из хранилища, она получает пустую коллекцию. Она содержит только события, возникшие в текущей транзакции.

Листинг 18.16. Сохранение коллекции событий в сущности

```
public class OrderForDelivery
{
    ...

    public List<Object> RecordedEvents { get; private set; }

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                RecordedEvents.Add(new DeliveryGuaranteeFailed(this));
            }
        }
    }

    ...
}
```

Когда вызывается метод `ConfirmReceipt`, вместо немедленной публикации события (как в листинге 18.15) он добавляет его в коллекцию `RecordedEvents` сущности `OrderForDelivery`, как показано в листинге 18.16. Обратите внимание, что коллекция общедоступна и, соответственно, доступ к ней можно получить за пределами предметной модели.

Любые побочные эффекты, возникающие при обработке событий, не возникают немедленно, как в случае, когда события публикуются сразу же. Некоторые считают такое поведение желательным, потому что побочные эффекты не влияют на выполнение текущего метода.

Когда сущность завершает выполнение своей логики, управление будет передано прикладной службе, реализующей сценарий использования. Именно в этот момент события из сущности могут быть переданы диспетчеру, как показано в листинге 18.17.

Листинг 18.17. Распространение событий в уровне служб

```
public class ConfirmDeliveryOfOrder
{
    private IOrderRepository orderRepository;
    private IEventDispatcher dispatcher;
```

```
public ConfirmDeliveryOfOrder(IOrderRepository orderRepository,
    IEventDispatcher dispatcher)
{
    this.orderRepository = orderRepository;
    this.dispatcher = dispatcher;
}

public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{
    var order = orderRepository.FindBy(orderId);
    order.ConfirmReceipt(timeThatPizzaWasDelivered);

    foreach (var evnt in order.RecordedEvents)
    {
        dispatcher.Dispatch(evnt);
    }
}
```

Если нежелательно передавать диспетчер событий в конструкторы предметных объектов, можно воспользоваться приемом внедрения в события. В листинге 18.18 показано, как диспетчер событий передается в метод `ConfirmReceipt()`, который записывает событие в диспетчер, а не в сущность. Отметим, что диспетчер не выполняет передачу события немедленно. Он сохраняет события и выполнит их отправку разом после получения команды из уровня служб, как было показано в листинге 18.17.

Листинг 18.18. Запись событий в диспетчер

```
public class OrderForDelivery
{
    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered,
        IEventDispatcher dispatcher)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                Dispatcher.Record(new DeliveryGuaranteeFailed(this));
            }
        }
    }

    ...
}
```

Понятно, что прием внедрения диспетчера в методы загрязняет сигнатуру предметной модели, поэтому, используя данный шаблон, следует проявлять особую осторожность. Чаще всего вы, вероятно, будете использовать прием сохранения событий в сущности, применяя подход на основе передачи диспетчера только тогда, когда выгоды от его применения будут достаточно очевидны.

Для реализации собственного диспетчера вы можете взять за основу практически весь листинг 18.11, за исключением момента объявления класса статическим. Также можно использовать контейнер IoC, как это показано в следующем разделе.

ПРИМЕЧАНИЕ

Сохранение событий в корне агрегата считается важнейшим аспектом создания предметных моделей с поддержкой регистрации событий. Поэтому если вы задумываетесь о регистрации событий, описанный выше подход прекрасно подойдет вам. Более подробную информацию вы найдете в главе 22.

Использование контейнера IoC в качестве диспетчера событий

При использовании контейнера IoC есть возможность наделить его функциями диспетчера событий. Это может оказаться неплохим вариантом, если вы уже используете контейнер IoC или не хотите создавать собственный диспетчер событий. Также этот подход может пригодиться, когда вам необходимы более широкие возможности контейнера IoC, включая управление жизненным циклом или регистрацию обработчиков на основе соглашений. Такая регистрация обработчиков на основе соглашений показана в листинге 18.19, где IoC-контейнер StructureMap сканирует все сборки в проекте и автоматически регистрирует каждый обработчик событий с соответствующим ему событием. Однако этот шаблон требует создания интерфейса, представляющего обработчики событий. В листинге 18.19 показан используемый интерфейс `IHandles<T>`.

Листинг 18.19. Автоматическая регистрация обработчиков с IoC-контейнером StructureMap

```
return new Container(x =>
{
    x.Scan(y =>
    {
        y.AssembliesFromApplicationBaseDirectory();
        y.AddAllTypesOf(typeof(IHandles<>));
        y.WithDefaultConventions();
    });
});
```

В листинге 18.20 показано, как после регистрации обработчиков событий можно использовать контейнер из прикладной службы для рассылки всех событий, записанных в ходе выполнения текущей транзакции.

Листинг 18.20. Использование IoC-контейнера для рассылки событий

```
public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{
    var order = orderRepository.FindBy(orderId);
    order.ConfirmReceipt(timeThatPizzaWasDelivered);

    foreach(var handler in Container.ResolveAll<IHandles<T>>())
        handler.Handle(order.RecordedEvents);
}
```

Тестирование предметных событий

Тестировать программный код, использующий шаблон «Предметные события», ничуть не сложнее тестирования обычного объектно-ориентированного кода. А иногда даже проще — из-за отсутствия необходимости писать фиктивные реализации сотрудничающих объектов. Однако порядок тестирования имеет свои особенности, поэтому в данном разделе дается краткое руководство с примерами.

Модульное тестирование

Чаше всего в модульных тестах требуется убедиться, что сущность или предметная служба возбуждает событие, чтобы сообщить о важном изменении. Продолжая сценарий с доставкой пиццы, это можно продемонстрировать на примере упомянутого ранее события `DeliveryGuaranteeFailed`. В листинге 18.21 представлен модульный тест, проверяющий возбуждение события с использованием статической версии `DomainEvents`.

Листинг 18.21. Модульный тест для сущности, возбуждающей предметные события

```
[TestClass]
public class Delivery_guarantee_events_are_raised_on_guarantee_offer_failure
{

    public bool eventWasRaised = false;

    [TestInitialize]
    public void When_confirming_an_order_that_is_late()
    {
        offer.Stub(x =>
            x.IsNotSatisfiedBy(timeOrderWasPlaced, timePizzaDelivered)
        ).Return(true);

        var order = new OrderForDelivery(id, customerId,
            restaurantId, menuItemIds, timeOrderWasPlaced, offer
        );

        using (DomainEvents.Register<DeliveryGuaranteeFailed>(setTestFlag))
```

```

        {
            order.ConfirmReceipt(timePizzaDelivered);
        }
    }

    private void setTestFlag(DeliveryGuaranteeFailed obj)
    {
        eventWasRaised = true;
    }

    [TestMethod]
    public void A_delivery_guarantee_failed_event_will_be_raised()
    {
        Assert.IsTrue(eventWasRaised);
    }
}

```

Статическая природа класса `DomainEvents` в действительности не усложняет тестирование, как можно видеть в листинге 18.21. При использовании шаблона «Предметные события» нет необходимости писать фиктивную реализацию класса `DomainEvents`, а это говорит, что тесная связь не является такой уж большой проблемой. Это означает, что с помощью модульного теста можно проверить возбуждение соответствующего события, зарегистрировав функцию обратного вызова, которая устанавливает флаг. В листинге 18.21 таким флагом является переменная `eventWasRaised`. Если `eventWasRaised` получает значение `true` по окончании теста, значит, событие было возбуждено.

Аналогично, при записи предметных событий снова вызывается желаемое предметное поведение, только вместо публикации события проверяется его запись, как показано в листинге 18.22.

Листинг 18.22. Модульный тест агрегата, возвращающего предметные события

```

[TestClass]
public class Delivery_guarantee_events_are_recorded_on_guarantee_offer_failure
{

    [TestInitialize]
    public void When_confirming_an_order_that_is_late()
    {
        offer.Stub(x =>
            x.IsNotSatisfiedBy(timeOrderWasPlaced, timePizzaDelivered)
        ).Return(true);

        order = new OrderForDelivery(id, customerId, restaurantId,
            menuItemIds, timeOrderWasPlaced, offer
        );

        order.ConfirmReceipt(timePizzaDelivered);
    }
}

```

```
[TestMethod]
public void A_delivery_guarantee_failed_event_will_be_recorded()
{
    var wasRecorded = order
        .RecordedEvents
        .OfType<DeliveryGuaranteeFailed>()
        .Count() == 1;

    Assert.IsTrue(wasRecorded);
}
```

Тестирование прикладной службы

Интеграционное тестирование в уровне прикладных служб также становится возможным при использовании предметных событий. Более того, ваши тесты, скорее всего, будут очень похожи на подобные тесты служб, не использующих предметные события. Это объясняется тем, что события и обработчики событий являются лишь техническими деталями реализации, как это демонстрирует пример в листинге 18.23.

Листинг 18.23. Тестирование прикладных служб при использовании предметных событий

```
[TestClass]
public class Delivery_guarantee_failed
{
    IBus bus = MockRepository.GenerateStub<IBus>();

    // при тестировании в уровне служб может потребоваться
    // использовать конкретный репозиторий
    IOrderRepository repo = MockRepository.GenerateStub<IOrderRepository>();
    Guid orderId = Guid.NewGuid();
    Guid customerId = Guid.NewGuid();
    Guid restaurantId = Guid.NewGuid();
    List<int> itemIds = new List<int> { 123, 456, 789 };
    DateTime timeOrderWasPlaced = new DateTime(2015, 03, 01, 20, 15, 0);

    [TestInitialize]
    public void If_an_order_is_not_delivered_within_the_agreed_upon_timeframe()
    {
        var offer = new ThirtyMinuteDeliveryGuaranteeOffer();

        // если прошло более 30 минут, значит, гарантии своевременной
        // доставки нарушены
        var timeOrderWasReceived = timeOrderWasPlaced.AddMinutes(31);

        var order = new OrderForDelivery(
            orderId, customerId, restaurantId, itemIds, timeOrderWasPlaced, offer
        );

        repo.Stub(r => r.FindBy(orderId)).Return(order);
    }
}
```

```
var service = new ConfirmDeliveryOfOrder(repo, bus);
service.Confirm(timeOrderWasReceived, orderId);
}

[TestMethod]
public void An_external_refund_due_to_late_delivery_instruction_will_be_
published()
{
    // получить первое сообщение,
    // опубликованное в ходе выполнения этого сценария
    var message = bus.GetArgumentsForCallsMadeOn(
        b => b.Send(new RefundDueToLateDelivery()),
        x => x.IgnoreArguments()
    )[0][0];

    var refund = message as RefundDueToLateDelivery;

    Assert.IsNotNull(refund);
    Assert.AreEqual(refund.OrderId, orderId);
}
}
```

Нигде в листинге 18.23 нет ни малейшего намека на шаблон «Предметные события». Тестовые данные и службы подготовлены, экземпляр прикладной службы `ConfirmDeliveryOfOrder` создается и вызывается, и, в заключение, выполняется проверка, опубликовал ли `NServiceBus` внешнее событие `RefundDueToLateDelivery` для обработки в другом ограниченном контексте. Этот тест имел бы точно такую же организацию, даже если бы реализация предметной модели не использовала шаблон «Предметные события». Этот пример показывает, что использование предметных событий не усложняет тестирование в уровне прикладных служб. Это также означает, что можно реорганизовать существующие предметные модели, дополнив их поддержкой предметных событий без изменения тестов для уровня прикладных служб. И они помогут вам убедиться, что вы ничего не нарушили в процессе такой реорганизации.

ПРИМЕЧАНИЕ

Прикладные службы более подробно рассматриваются в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования», включая приемы и стратегии тестирования.

Ключевые идеи

- Предметные события соответствуют наиболее важным происшествиям в реалиях предметной области; они являются частью единого языка.
- Предметные события — это также название шаблона проектирования, помогающего более явно выразить предметные события в программном коде.

- Шаблон «Предметные события» напоминает шаблон «Издатель/подписчик», где издатели возбуждают события, а подписчики обрабатывают их.
- В отличие от сообщений, которыми обмениваются ограниченные контексты (см. главы с 11-й по 13-ю), предметные события курсируют в рамках одной предметной модели и обычно синхронны.
- Использование предметных событий может упростить перевод некоторых операций или сценариев использования на асинхронную обработку.
- Обработчиками событий могут быть классы или анонимные функции/лямбда-выражения.
- Некоторые обработчики находятся в предметной модели, а некоторые — в уровне служб.
- Обработчики в предметной модели выполняют предметную логику.
- Обработчики в уровне предметных служб обычно отвечают за решение некоторых задач, свойственных прикладным службам.
- Существует множество версий шаблона «Предметные события».
- Один из подходов к реализации шаблона «Предметные события» — использовать встроенные конструкции языка, такие как ключевое слово `event` в C#.
- Встроенная поддержка событий в C# способствует образованию тесных связей между издателями и подписчиками. Для ослабления связей можно использовать шину памяти, такую как `NServiceBus`.
- Шина памяти обеспечивает непосредственную реакцию на события и соответствующие им побочные эффекты. Если это нежелательно, можно организовать сохранение событий и их передачу для обработки с помощью диспетчера.
- Еще одна версия шаблона опирается на статический класс `DomainEvents`, напоминающий шину сообщений в памяти, но намного более компактный.
- Использование предметных событий не означает необходимости употребления механизмов регистрации событий, хотя многие разработчики находят такую комбинацию весьма эффективной.
- Самое главное: шаблон «Предметные события» является неким компромиссом — дополнительным уровнем, который распределяет программный код по большему числу файлов, но также помогает получить полностью инкапсулированную предметную модель, ставящую акценты на события, фактически протекающие в предметной области.

19

Агрегаты

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Реализация сложных отношений между сущностями и объектами-значениями
- Почему агрегаты считаются самым мощным тактическим шаблоном
- Как определять и структурировать агрегаты
- Роль корня агрегата
- Принципы согласованности агрегатов

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.worx.com/go/domaindrivendesign на вкладке **Download Code** (Загружаемый код). Примеры кода для главы 19 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Структура предметной модели определяется множеством сущностей и объектов-значений, представляющих понятия предметной области. Однако множество и разнообразие отношений между предметными объектами могут усложнить реализацию предметной модели в программном коде и сделать его запутанным. Связи между объектами, не имеющие поведения и существующие, только чтобы лучше отразить реальность, привносят в предметную модель ненужную сложность. Двухнаправленные связи также безосновательно увеличивают сложность реализации. Именно поэтому проектирование отношений между предметными объектами является таким же важным аспектом разработки, как и проектирование самих предметных объектов.

Даже в тех случаях, когда все связи в модели имеют строгое обоснование, большие модели все еще влекут за собой множество технических сложностей, мешающих определять границы транзакций и согласованности, которые соответствовали бы предметной области и обеспечивали эффективность работы.

Эта глава охватывает наиболее удачные приемы, помогающие сохранить отношения между предметными объектами простыми и соответствующими инвариантам предметной области. Здесь также будет представлено предметно-ориенти-

рованное понятие агрегата (aggregate) — границы согласованности (consistency boundary), помогающей разделить большие модели на небольшие группы предметных объектов, которыми технически проще управлять. Цель обсуждения этих тем — помочь вам управлять сложностью своих предметных моделей.

Агрегаты — чрезвычайно важный шаблон проектирования предметных моделей. Они помогают управлять технической сложностью и добавляют еще один уровень абстракции, помогающий упростить рассуждения о предметной модели.

Управление сложными деревьями объектов

На ранних этапах проектирования модели начинающие практики DDD обычно склонны сосредоточиваться на сущностях и объектах-значениях, уделяя слишком мало внимания отношениям между ними. Они часто конструируют связи, отражающие реальный мир, или, что еще хуже, лежащую в основе модель данных. В результате необоснованного объединения объектов и несоответствия их инвариантам предметной области получается предметная модель, скрывающая важнейшие понятия, запутывающая специалистов и разработчиков и технически сложная в реализации.

Зависимости могут стать трудноразрешимыми, особенно когда имеются отношения «многие ко многим». Важно постоянно напоминать себе, что модель предметной области — это далеко не то же самое, что модель данных, и одной из важнейших, пожалуй, целей предметной модели является поддержка инвариантов и сценариев использования, а не пользовательских интерфейсов.

Избежать создания сложных деревьев объектов совсем нетрудно. Число связей между сущностями и объектами-значениями легко можно уменьшить, ограничившись однонаправленными взаимоотношениями. Для сокращения числа отношений можно прибегнуть к обоснованию их включения. Если отношение не является строго необходимым (не используется для удовлетворения инварианта), откажитесь от его реализации. Вообще старайтесь не моделировать реальность. Существование отношений в предметной области допускается, но их реализация в программном коде может не нести никакой выгоды.

Говоря более простым языком, если моделировать связи между предметными объектами исходя из потребностей сценариев использования, а не в попытке отразить действительность, то это поможет упростить предметную модель и создать более эффективную систему. Связи между предметными объектами должны поддерживать инварианты, а не решать проблемы, имеющие отношение к пользовательскому интерфейсу. Упрощайте связи в деревьях объектов. Моделируйте однонаправленные взаимоотношения. Упростив модель, вы обеспечите простоту ее реализации и последующего сопровождения.

Однонаправленность предпочтительнее

Модель, отражающая действительность, будет содержать множество двунаправленных связей между объектами, когда пары объектов ссылаются друг на друга. Возьмем для примера систему закупок, изображенную на рис. 19.1. Она позволяет перемещаться между поставщиками (Supplier) и списками заказов

(PurchaseOrder) в обоих направлениях. Аналогично имеется возможность перемещаться между поставщиками (Supplier) и всеми продуктами (Product), которые они поставляют, также в обоих направлениях. Более того, можно даже перемещаться в обоих направлениях между заказами (PurchaseOrder) и соответствующими списками продуктов (Product). Инструменты объектно-реляционных отображений (ORM) облегчают определение таких двунаправленных отношений в программном коде, но это может приводить к образованию чересчур глубоких деревьев объектов и к снижению производительности.

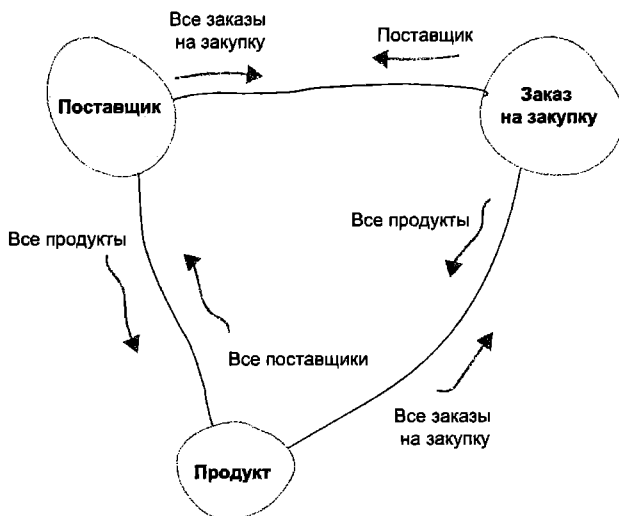


Рис. 19.1. Двунаправленные отношения увеличивают сложность модели

ПРИМЕЧАНИЕ

Иногда двунаправленные отношения могут казаться необходимыми для составления отчетов. Однако отчеты часто можно рассматривать как отдельную разновидность задач, для реализации которых используются специализированные шаблоны. Соответствующие примеры можно найти в главе 26 «Запросы: предметная отчетность».

Листинг 19.1. Реализация отношений между объектами

```
public class Supplier
{
    ...

    public IList<PurchaseOrder> PurchaseOrders { get; private set; }
    public IList<Product> Products { get; private set; }

    ...
}

public class PurchaseOrder
{

```



```
public Supplier Supplier {get; private set;}
public IList <Product> Products {get; private set;}

...
}

public class Product
{
    ...

    public IList<Supplier> Suppliers {get; private set;}
    public IList<PurchaseOrder> PurchaseOrders {get; private set;}

    ...
}
```

Но так ли необходимо загружать все объекты *Product*, если требуется всего лишь изменить контактную информацию в объекте *Supplier*? Имеет ли смысл хранить в объекте *Product* список *Suppliers* всех поставщиков? Как видите, двунаправленные отношения увеличивают сложность реализации и затеняют предметные понятия. Особенно важное предметное понятие, которое здесь скрывается, — это владелец отношения. Чтобы упростить отношения, можно ограничиться единственным направлением обхода, как показано на рис. 19.2.

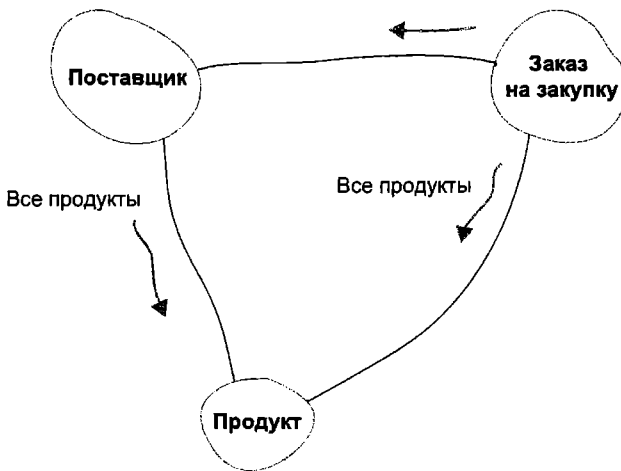


Рис. 19.2. Ограничение числа двунаправленных отношений

Благодаря выбору однонаправленных отношений между сущностями, удалось упростить предметную модель. По сути, двунаправленные отношения были просто преобразованы в однонаправленные. Это основной шаблон уменьшения сложности, который вы свободно можете использовать.

Определяя отношения между объектами, всегда задавайте себе вопрос: «Какие функции будет выполнять эта связь и кому она нужна для работы?». Это поможет вам избежать создания ненужных двунаправленных отношений. Сценарий заку-

пок — слишком простой пример, но в больших предметных моделях с множеством двунаправленных связей сложность может возрасти невероятно быстро, особенно когда дело доходит до сохранения и извлечения объектов.

Ограничение связей

Если связи реализуются как ссылки между объектами для поддержки предметных инвариантов и имеют вид «один ко многим» или «многие ко многим», их необходимо ограничивать так, чтобы уменьшить число объектов, которые требуется извлекать. Взгляните на рис. 19.3. Предметный объект `Contract` представляет договор на предоставление услуг мобильной связи, а `Calls` — все звонки, выполненные в соответствии с этим договором. Объект `Calls` должен знать, сколько неизрасходованных минут осталось в текущем периоде, чтобы обеспечить корректный учет. На двадцать третьем месяце 24-месячного договора для клиента накопится информация о сотнях звонков. Загрузка всех объектов `Calls` ухудшит производительность. Однако в этом нет никакой необходимости, если все звонки нужны инварианту только для того, чтобы определить число неизрасходованных минут. Связь следует ограничить так, чтобы извлекались только звонки за текущий период, как показано на рис. 19.4.

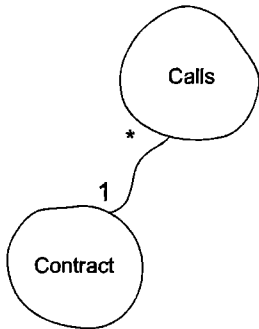


Рис. 19.3. Ограничение связей

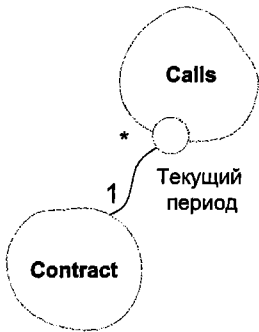


Рис. 19.4. Ограничение связей критерием фильтрации, входящим в состав единого языка

Для оптимизации производительности такое определение можно применить на уровне доступа к данным/базы данных, как показано в листинге 19.2.

Листинг 19.2. Применение определения связи на уровне хранилища данных

```
public class ContractRepository
{
    public Contract GetBy(ContractId id, Period period)
    {
        ....
    }
}
```

Вообще говоря, чем большее число элементов содержит связанная коллекция (это определяет ее *мощность*), тем выше сложность технической реализации. Поэтому старайтесь уменьшать мощность коллекций, добавляя ограничения. Работая с коллекциями, лучше стремиться к четкому определению операций; избегайте попадания в ловушку простого воспроизведения модели данных в программном коде.

Идентификаторы объектов предпочтительнее ссылок

Как не раз повторялось в этой книге, главной целью предметной модели является моделирование инвариантов для поддержки сценариев использования. Поэтому между предметными объектами должны существовать только отношения, отвечающие функциональным потребностям. Отношения, не имеющие поведения, лишь усложняют реализацию предметной модели. Классическим примером ненужного дополнения являются ссылки между объектами, усложняющие отношения в предметной модели.

По вполне понятным причинам многие разработчики считают естественным моделирование отношений в программном коде в виде ссылок между объектами. Например, в реальной жизни клиент может оформить множество заказов, но в пространстве решений приложения может не быть инварианта, требующего, чтобы объект *Customer*, представляющий клиента, хранил коллекцию со всеми заказами *Order*, принадлежащими ему. Моделирование такого отношения с применением ссылок между объектами, как показано на рис. 19.5, добавляет ненужные сложности.

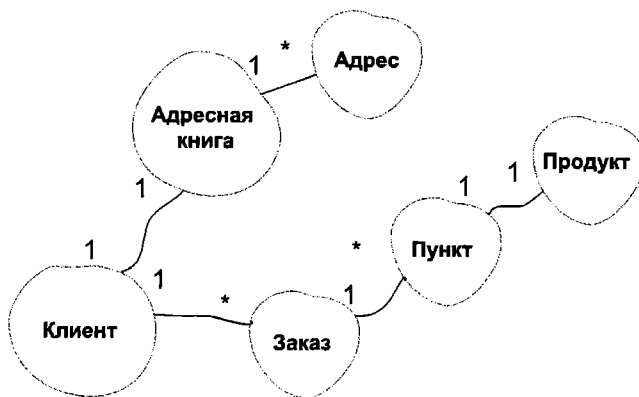


Рис. 19.5. Моделирование отношений с применением ссылок между объектами увеличивает сложность

В связи с тем, что каждая связь реализуется как ссылка между объектами, для размещения заказа необходимо использовать следующий код, позволяющий совершить обход дерева объектов.

Листинг 19.3. Размещение заказа со связями, реализованными в виде ссылок между объектами

```
public class OrderApplicationService
{
    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);

        customer.AddOrder(products);
    }
}
```

Этот код может работать очень медленно, потому что для выполнения простого сценария размещения заказа загружается большое дерево объектов (объект `Customer` со всеми имеющимися объектами `Order`). В действительности все, что требуется в данной ситуации, — это извлечь объект, представляющий клиента, и добавить новый заказ в коллекцию его заказов, указав адрес доставки по умолчанию, извлеченный из объекта клиента. А множество объектов, загружаемых в этом сценарии, вообще никак не используются. Эти проблемы возникают из-за того, что отношения между объектами смоделированы в виде ссылок и требуют загрузить все дерево объектов из хранилища. Кто-то из вас может заметить, что отложенная загрузка устраняет эти проблемы, но все дело в том, что отложенная загрузка сама по себе также может усложнить модель; кроме того, такой подход не особенно четко отражает порядок использования предметных объектов для выполнения сценария использования.

Альтернативный метод реализации связей заключается в сохранении идентификатора объекта и использовании репозитория внутри прикладной службы для извлечения предметных объектов, действительно необходимых для выполнения сценария. Используя репозиторий, можно избавиться от ссылок между объектами в модели и тем самым от ненужных сложностей. На рис. 19.6 изображена более ясная модель объектов, без ссылок между ними.

Используя идентификаторы и репозитории, необходимо вынести координацию на уровень прикладных служб, как показано в листинге 19.4. В этом случае вызов `_addressBookRepository.FindBy()` является альтернативой использованию ссылки в объекте `Customer`.



Рис. 19.6. Упрощение отношений с использованием идентификаторов вместо ссылок

Листинг 19.4. Размещение заказа в случае реализации связей между объектами только с применением идентификаторов

```
public class OrderApplicationService
{
    ...

    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);

        var addressBook = _addressBookRepository.FindBy(customerId);

        var order = customer.CreateOrderFor(products);

        order.DispatchTo(addressBook.Default);

        _orderRepository.Add(order);
    }
}
```

Определить необходимость ссылки на объект можно, задав простой вопрос: «Предназначена ли данная связь для поддержки предметного инварианта в особых сценариях использования?». В примере с клиентами и заказами клиент владеет заказами, но для удовлетворения инвариантов заказа необходим только идентификатор клиента; ему не нужна прямая ссылка на объект, представляющий клиента. Не стремитесь создавать модели, в точности соответствующие реальности, или добавлять ссылки на объекты, чтобы обеспечить соответствие модели данных; добавляйте ссылки на объекты, только когда они действительно необходимы для удовлетворения требований инвариантов. Во всех прочих случаях следует отдать предпочтение идентификаторам и репозиториям, чтобы ослабить тесные связи, образующиеся в предметной модели.

На рис. 19.7 изображена общая структура предметной модели, с ненужной точностью отражающей реальность и содержащей множество ссылок между объектами. На рис. 19.8 изображена альтернативная, фрагментированная модель с небольшим числом связей между объектами, где вместо ссылок используются идентификаторы. Если вы последуете совету в этом разделе, изображение на рис. 19.8 будет для вас целью, к которой следует стремиться для получения ясной и технически простой предметной модели. В оставшейся части главы вы увидите, как агрегаты DDD помогают достичь этой цели.

Агрегаты

Сокращение и ограничение отношений между предметными объектами упрощает техническую реализацию и отражает глубокое понимание предметной области. Весьма желательно максимально упрощать программный код, чтобы облегчить управление понятиями и их обсуждение. Тем не менее сохраняется необходимость объединения объектов, используемых совместно, для повышения эффективности

и надежности системы. Агрегаты помогают достичь всех этих целей за счет образования связанных групп объектов вокруг предметных инвариантов, которые действуют согласованно и одновременно.

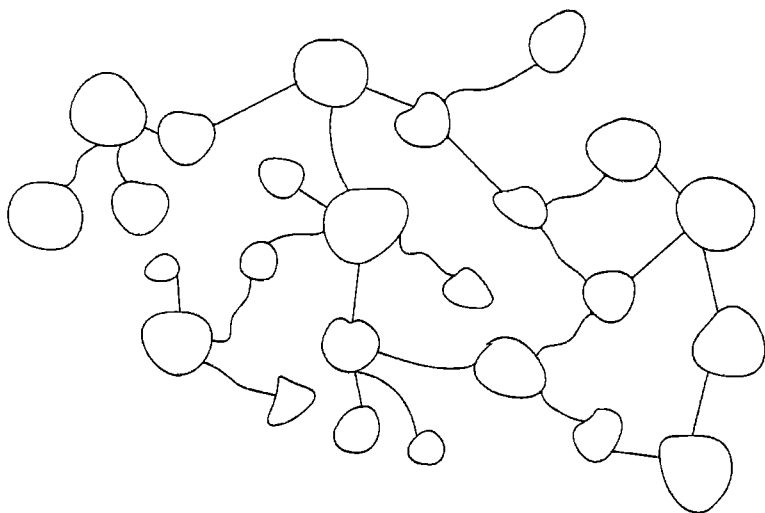


Рис. 19.7. Сложная предметная модель с множеством ненужных связей

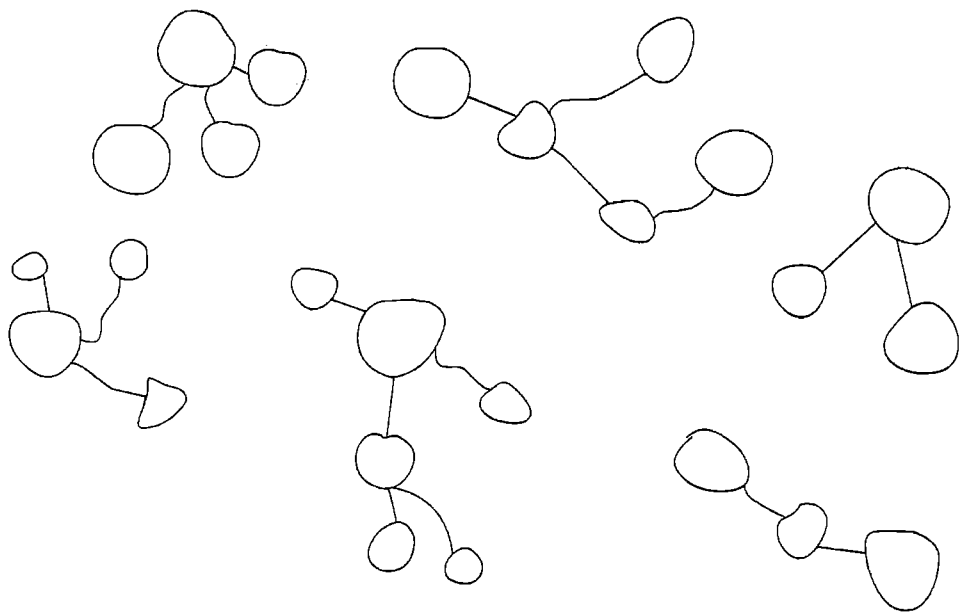


Рис. 19.8. Простая и ясная предметная модель, включающая только действительно необходимые связи

Агрегаты являются одним из самых мощных тактических шаблонов, но также и самым сложным для понимания. Существует множество принципов и рекомендаций, помогающих в создании эффективных агрегатов, но часто разработчики все свое внимание сосредоточивают только на реализации этих правил, упуская из виду истинное назначение агрегатов — обеспечение границ согласованности.

Проектирование на основе предметных инвариантов

Предметные инварианты (domain invariants) — это инструкции, или правила, которых всегда следует придерживаться. Нарушение предметных инвариантов повлечет неточности в моделировании предметной области. Простым примером инвариантов может служить следующее правило: победившая ставка на аукционе всегда должна предшествовать окончанию торгов. Если победившая ставка будет сделана после окончания аукциона, предметная модель окажется в недопустимом состоянии, потому что инвариант был нарушен и предметной модели не удалось правильно применить предметные правила.

Для проектирования агрегатов определенно имеет смысл использовать инварианты, потому что инварианты часто связаны с множеством предметных объектов. Когда все предметные объекты, связанные с инвариантом, находятся внутри одного агрегата, их проще сравнивать и согласовывать и тем самым гарантировать, что их коллективное состояние не нарушает предметный инвариант.

В сценарии с победившей ставкой аукцион и ставка могут быть смоделированы как разные объекты. Если оба объекта окажутся в разных агрегатах, тогда никакой единственный агрегат не сможет сослаться на оба объекта и гарантировать, что ни в какой момент инвариант не был нарушен даже на мгновение. Это довольно плохо, потому что инварианты никогда не должны нарушаться.

ПРИМЕЧАНИЕ

Вы уже встречались с инвариантами в главе 15 «Объекты-значения». Несмотря на близкое сходство, инварианты в той главе применялись к единственной сущности, тогда как в этой главе обсуждаются инварианты, которые часто применяются к комбинации предметных объектов.

Более высокий уровень абстракции предметной области

Группируя связанные предметные объекты, вы получаете возможность ссылаться на них как на единое понятие, что весьма способствует более эффективному общению и рассуждениям. Агрегаты приносят эти преимущества абстракций в предметную модель, позволяя ссылаться на коллекции предметных объектов как на единое понятие. Например, вместо того чтобы ссылаться отдельно на «заказ» и «очереди заказов», вы можете сгруппировать их в единое понятие, например «заказ», и сослаться уже на него.

Границы согласованности

Чтобы гарантировать пригодность и надежность системы, совершенно необходимо правильно определить, какие данные должны быть согласованы и где должны пролегать границы транзакций. При применении DDD все это определяется группировкой объектов, вовлеченных в реализацию одного и того же сценария использования. Такие согласованные группы предметных объектов называют агрегатами.

Строгая согласованность внутри

Один из вариантов поддержки согласованности — создать единственный агрегат, заключив всю предметную модель в границы единой транзакции. Проблема такого подхода состоит в том, что в многопользовательских системах, где одновременно выполняется множество изменений, есть вероятность конфликта изменений, которые совершенно не пересекаются. Вероятнее всего, данная проблема проявлялась бы как блокировка доступа на уровне базы данных или отказ применить изменения (из-за пессимистического управления одновременным доступом).

На рис. 19.9 демонстрируются проблемы, которые могут возникнуть в результате неоптимального определения границ транзакции, охватывающих слишком большое число объектов. Пользователь А желает добавить адрес в запись о клиенте, а пользователь Б — изменить состояние заказа. Здесь нет никаких инвариантов, требующих, чтобы во время изменения параметров заказа персональные данные клиента оставались в неприкосновенности, но если в этом сценарии попробовать одновременно выполнить оба изменения, одна из попыток изменения будет заблокирована или отвергнута.

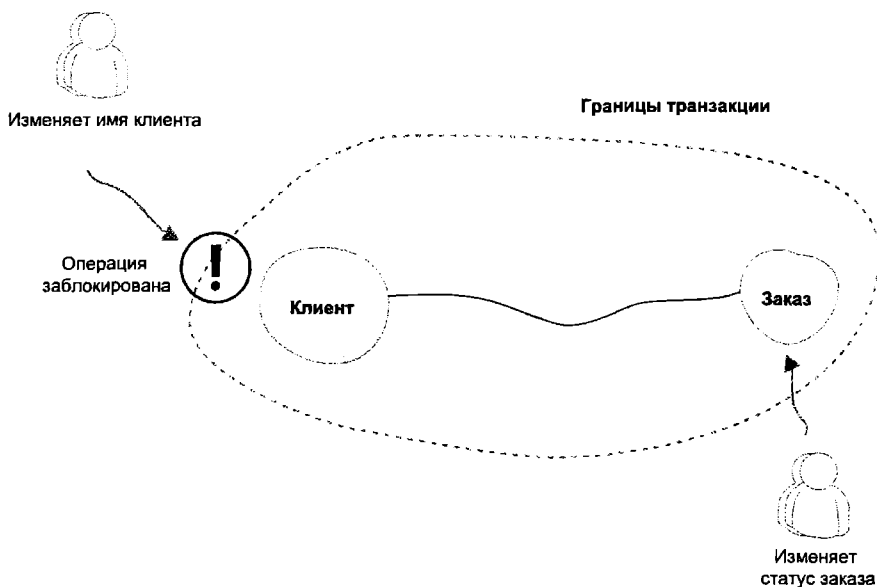


Рис. 19.9. Блокировка, возникающая из-за слишком обширных границ транзакции

Можно было бы предположить, что избавление от границ транзакции решит проблемы. Однако это опасное заблуждение, потому что за этим предположением последуют проблемы согласованности, вызванные нарушениями предметных инвариантов. На рис. 19.10 показано, как два разных пользователя редактируют один и тот же заказ; первый применяет код скидки к заказу, а второй — изменяет очередь заказа. Так как в данном случае нет никакого разграничения доступа из-за отсутствия транзакции, оба изменения будут приняты, что приведет к нарушению инварианта. В данном сценарии заказчик мог бы незаслуженно получить значительную скидку.

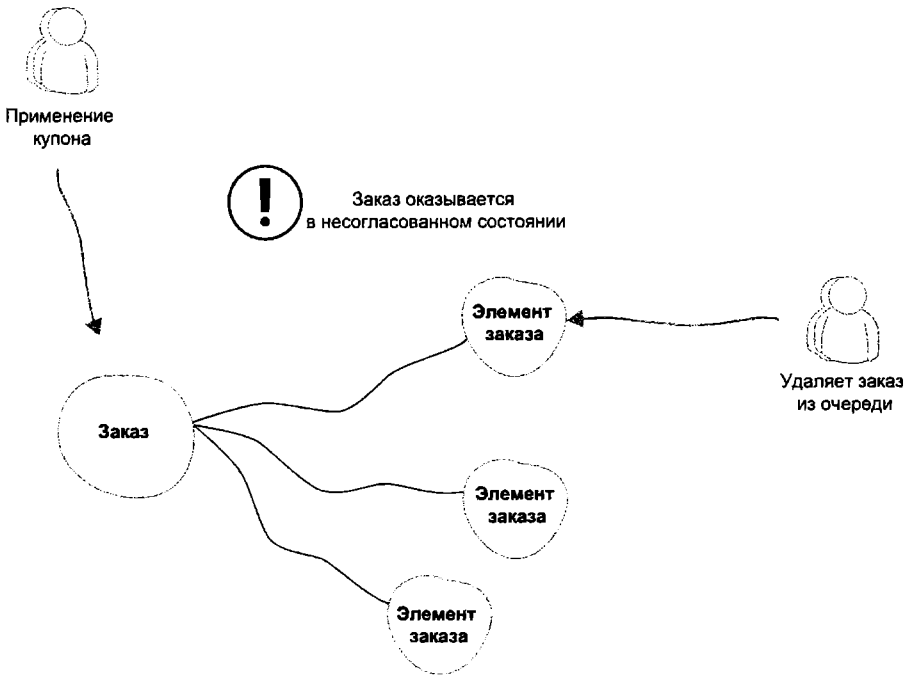


Рис. 19.10. Из-за отсутствия границ транзакции возникает несогласованность данных

Чтобы правильно определить границы согласованности и транзакций, практики DDD опираются на шаблон «Агрегат». Как уже отмечалось, *агрегат* — это группа предметных объектов, явно определяемая для поддержки поведения и инвариантов предметной модели, которая устанавливает границы транзакций и согласованности. Агрегат интерпретирует группу предметных объектов как единое целое: например, нет никаких очередей заказа — есть только заказ. Очереди заказа отсутствуют или не имеют смысла за пределами понятия заказа. Агрегат определяет границы группы предметных объектов и отделяет ее, в терминах согласованности и механизма транзакций, от всех остальных предметных объектов за пределами этой группы.

На рис. 19.11 изображен результат создания агрегата на основе инварианта в примере с заказом. Как можно видеть, теперь «клиент» и «заказ» интерпретируются как два независимых агрегата, потому что в предметной области нет ни одного инварианта, вовлекающего оба этих понятия.

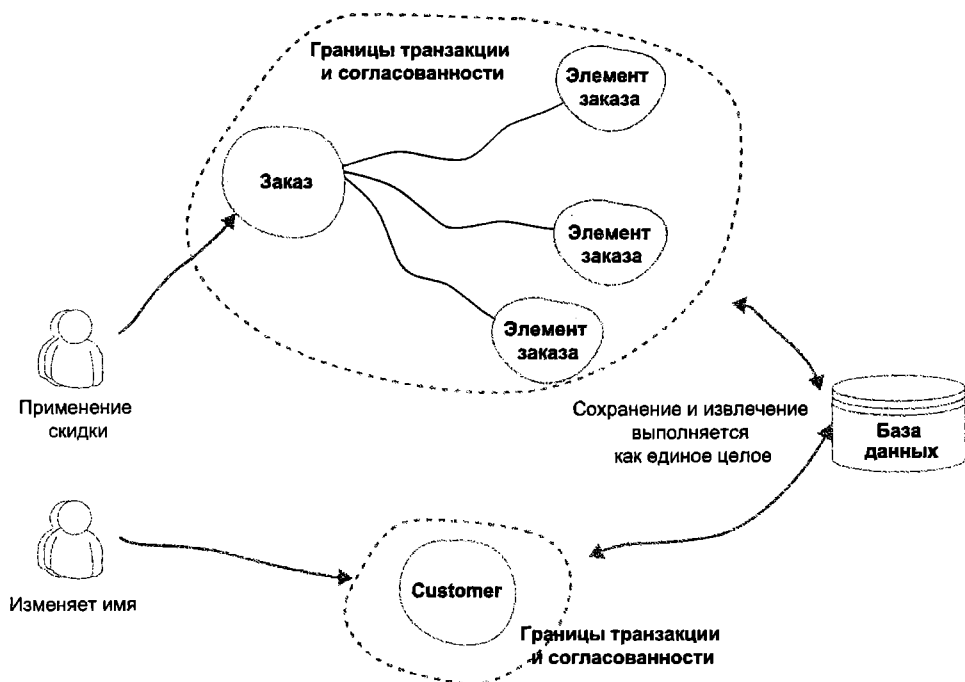


Рис. 19.11. Установка границ транзакций в соответствии с инвариантами

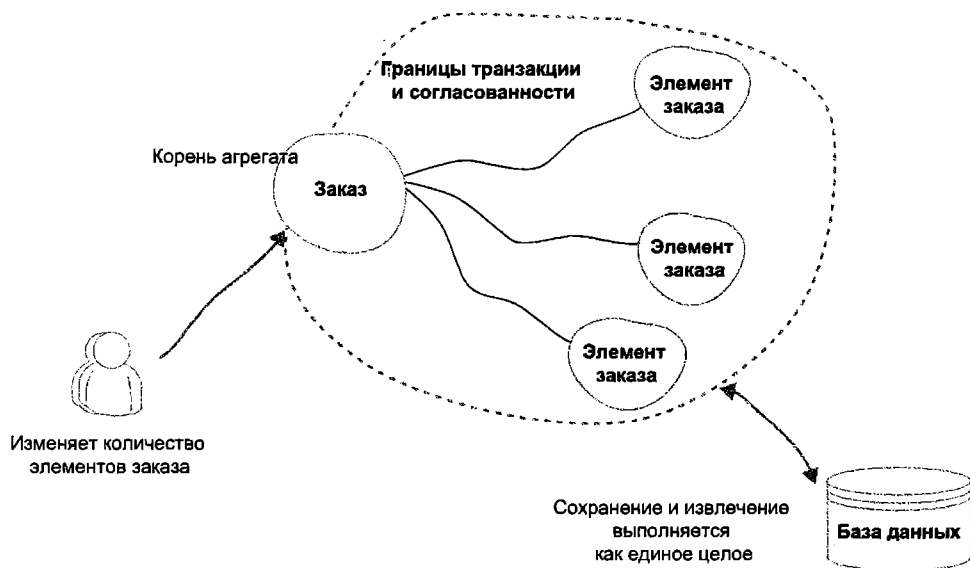


Рис. 19.12. Поддержание согласованности с помощью корней агрегатов

Чтобы обеспечить согласованность в группе предметных объектов, все операции с агрегатом должны производиться посредством единственной сущности, известной как корень агрегата (*aggregate root* — подробнее описывается далее в этой главе) и выделенной на рис. 19.12. Объекты за границами агрегатов не должны иметь прямого доступа к внутренним объектам агрегатов; это правило обеспечивает управляемость предметных объектов и гарантирует согласованность внутри агрегатов.

Предметные объекты не сохраняются и не извлекаются по отдельности. Извлечение агрегата из хранилища и запись его в хранилище выполняется целиком. Агрегаты — единственное, что может сохраняться в базу данных и извлекаться из нее. Никакие части агрегатов не могут сохраняться или извлекаться отдельно, если только они не используются для нужд составления отчетов. Это обеспечивает потенциальную согласованность между агрегатами.

Потенциальная согласованность снаружи

В связи с тем что сохранение и извлечение агрегатов выполняется атомарно, как единое целое, правило, распространяющееся на два или более агрегата, не будет согласованным немедленно (если хотя бы один агрегат в транзакции изменится). Вместо этого правило будет характеризоваться как потенциально согласованное. Это объясняется тем, что изменяемый агрегат гарантирует строгую согласованность только внутри. Он не отвечает за изменение чего бы то ни было за границами его соответствия. Поэтому агрегаты иногда могут хранить не самую свежую информацию, полученную ими из других агрегатов, как показано на рис. 19.13.

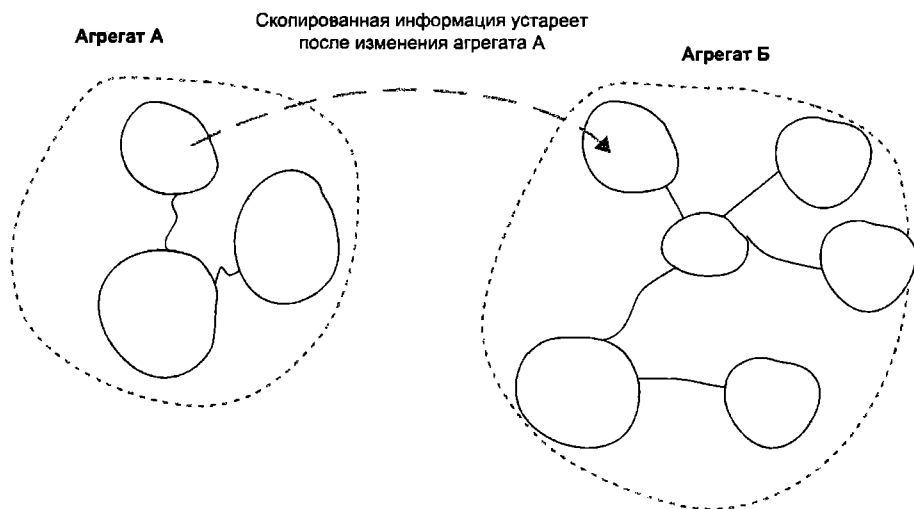


Рис. 19.13. Агрегаты потенциально согласованы снаружи

Для демонстрации правила потенциальной согласованности, которое распространяется на несколько агрегатов, рассмотрим правило лояльности: если за послед-

ний год клиент потратил более \$100, он получает 10% скидки на все последующие покупки. В предметной модели существуют отдельные агрегаты, представляющие заказы (Order) и лояльность (Loyalty). Когда размещается новый заказ, агрегат Order изменяется в рамках транзакции. В этот момент агрегат Loyalty имеет несогласованное представление об истории покупок, совершенных клиентом, потому что не обновляется в той же транзакции. Однако агрегат Order может публиковать событие, извещающее о создании заказа, на которое может быть подписан агрегат Loyalty. Когда объект, представляющий лояльность, получит событие, он сможет изменить оценку лояльности клиента, как показано на рис. 19.14.

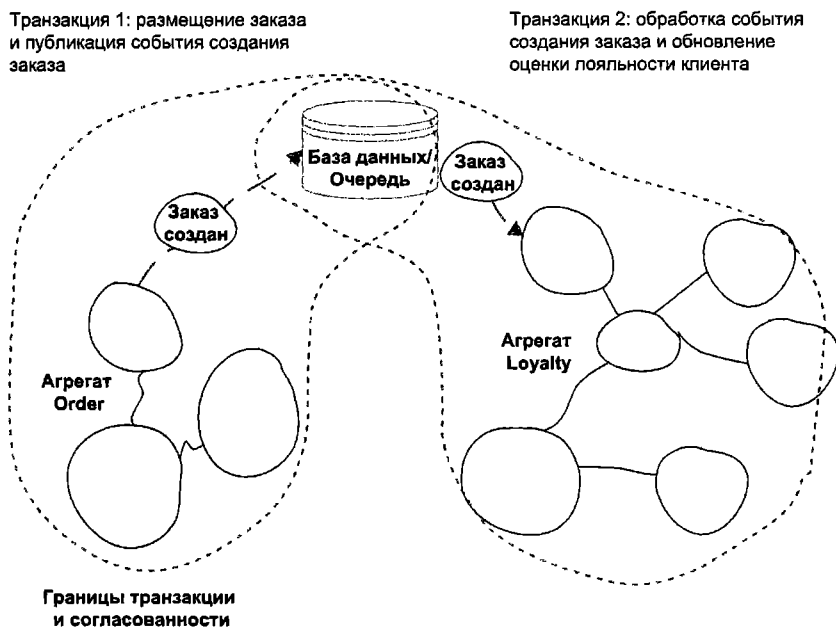


Рис. 19.14. Потенциальная согласованность агрегата Loyalty

Важно отметить, что в течение некоторого периода времени оценка лояльности будет неточно отражать сумму, потраченную клиентом. Само по себе это явление нежелательно, но оно является неплохим компромиссом, избавляющим от необходимости сохранять большое число объектов в рамках одной транзакции.

Очень важно довести до сведения заинтересованных лиц, что потенциальная согласованность агрегатов применяется с целью обеспечить удобство использования, однако в пограничных случаях это может приводить к рассогласованию агрегатов.

ПРИМЕЧАНИЕ

Шаблоны реализации потенциальной согласованности будут представлены далее в этой главе.

Специальные случаи

В действительности иногда бывает необходимо организовать изменение сразу нескольких агрегатов в рамках одной транзакции. Однако при этом важно понимать подоплеку рекомендаций и последствия их игнорирования.

Когда потенциальная согласованность обходится слишком дорого, вполне можно рассмотреть возможность изменения двух объектов в рамках одной и той же транзакции. К исключительным ситуациям можно отнести, например, тот случай, когда заинтересованные лица сообщают вам, что клиент может быть очень огорчен. Однако вы не должны слепо принимать к исполнению все пожелания заинтересованных лиц; они всегда будут с опаской относиться к потенциальной согласованности. Вы должны конкретизировать важность масштабируемости, производительности и других аспектов, на которые будет оказывать влияние отказ от использования потенциальной согласованности, чтобы заинтересованные лица могли принять обоснованное решение не в ущерб клиентам.

Иногда, когда сложность реализации потенциальной согласованности оказывается слишком большой, допускается отказаться от ее использования. Далее в этой главе вы увидите на примерах, что надежные реализации потенциальной согласованности часто основаны на асинхронных механизмах, добавляющих дополнительные сложности и зависимости.

Резюмируя вышесказанное: сохранение по одному агрегату в транзакции — это подход, используемый по умолчанию, но вы должны, в сотрудничестве с заинтересованными лицами, оценить техническую сложность каждого случая использования и сознательно игнорировать рекомендации, если существуют важные преимущества, такие как более полное удовлетворение потребностей пользователей.

ПРИМЕЧАНИЕ

Старайтесь не путать эту рекомендацию с загрузкой или созданием агрегатов. Вполне допустимо загружать множество агрегатов в рамках одной транзакции, при условии, что они будут сохраняться строго по одному. Точно так же в рамках одной транзакции вполне допустимо создавать множество агрегатов, потому что создание новых агрегатов не влечет за собой проблем параллельного доступа.

Маленькие агрегаты предпочтительнее

В общем случае чем меньше размер агрегата, тем быстрее и надежнее действует система, так как объем передаваемых данных уменьшается, вследствие чего снижается вероятность возникновения конфликтов параллельных операций. Соответственно следует стремиться создавать маленькие агрегаты. Попробуйте сначала определить маленький агрегат, а потом постепенно дополнять его новыми понятиями. Однако не забывайте, что важнейшим фактором остается точность предметной модели, поэтому размер агрегата не является определяющим критерием удачной архитектуры.

Далее обсуждаются некоторые следствия больших агрегатов, чтобы вы могли осознанно подходить к их проектированию, а не пытались слепо сделать их как можно

меньше. Иногда выгоднее оказывается использовать большие агрегаты, поэтому полезно знать, когда имеет смысл следовать или не следовать данной рекомендации.

Большие агрегаты могут ухудшать производительность

Каждый новый член агрегата увеличивает объем данных, который необходимо загружать из базы данных и сохранять в ней, а это напрямую влияет на производительность. Производительность может существенно падать, когда агрегат охватывает несколько таблиц или документов в базе данных. Для каждой таблицы требуется выполнить отдельный запрос или соединение, что определенно может ухудшить производительность локального процесса и увеличить нагрузку на серверы баз данных.

По общему признанию, затраты, связанные с большими агрегатами, во многих случаях оказываются незначительными. Но иногда производительность может падать до совершенно неприемлемого уровня. С уменьшением агрегатов снижается риск появления проблем производительности. Если впоследствии вам понадобится оптимизировать производительность, использование агрегатов маленького размера поможет вам быстрее найти компромиссное решение.

ВНИМАНИЕ

Уменьшение агрегатов исключительно для увеличения производительности системы считается вредной практикой. Обязательно исследуйте скорость обмена данными и производительность базы данных, чтобы глубже понять, что влияет на эффективность доступа к данным.

Большие агрегаты более восприимчивы к конфликтам одновременного доступа

Большой агрегат почти наверняка будет наделен более чем одной обязанностью и вовлечен в реализацию нескольких сценариев использования. Как следствие, увеличивается вероятность, что один и тот же агрегат попытаются изменить сразу несколько пользователей. А это, в свою очередь, увеличивает вероятность конфликтов, ухудшает удобство использования и порождает чувство неудовлетворенности у пользователей.

Это знание может пригодиться при проектировании агрегатов. Например, можно подсчитать число сценариев, в которых используется агрегат. Чем выше это число, тем больше сомнений должно быть в правильности границ агрегата и больше поводов поэкспериментировать с альтернативными решениями. Однако не забывайте о важности соответствия модели предметной области. Иногда оптимальным является использование агрегата в одном сценарии, а иногда некоторые агрегаты выгоднее использовать в нескольких сценариях.

Большие агрегаты могут плохо масштабироваться

При проектировании агрегатов необходимо также учитывать аспекты масштабируемости. Большие агрегаты могут храниться в базе данных в нескольких таблицах или документах. Это приводит к образованию тесных связей на уровне

базы данных, что может препятствовать перемещению или реорганизации подмножеств данных и отрицательно сказываться на масштабируемости.

Предметно-ориентированные агрегаты уменьшают число зависимостей между данными. Они дают возможность выполнять перемещение или реорганизацию данных на более детальном уровне. Например, если в системе электронной коммерции понадобится перенести информацию о заказах в другую базу данных, вы столкнетесь с большими сложностями, если потребуется реорганизовать агрегаты так, чтобы выделить информацию о заказах, без всякой надобности объединенную с другими данными, такими как адреса клиента или оценка лояльности.

В Интернете можно найти много историй о том, как разные компании перемещали часть своих данных в другие базы данных. Подобная практика использования нескольких баз данных для хранения разной информации получила название *polyglot persistence* (многовариантное хранение). Одним из известных примеров является попытка Британской вещательной корпорации Sky перенести хранение выходных данных из MySQL в Cassandra (<http://www.computerworlduk.com/it-vendors/sky-swaps-oracle-for-cassandra-reduce-online-shopping-errors-3474411/>) в целях устранить серьезное падение производительности с увеличением числа обслуживаемых клиентов.

Часто каждый ограниченный контекст получает свое собственное хранилище данных (см. главу 11 «Введение в интеграцию ограниченных контекстов»), что способствует улучшению масштабируемости. Но иногда узкое место может находиться внутри ограниченного контекста, и тогда маленькие агрегаты могут дать вам дополнительные возможности борьбы с ним.

Определение границ агрегатов

Граница агрегата определяет, какие объекты связаны друг с другом и какие предметные инварианты они будут претворять в жизнь. Это, возможно, самый важный аспект проектирования агрегатов. Данный раздел содержит описание множества принципов, которые помогут вам решить, какие объекты должны быть сгруппированы в агрегаты.

eBidder: интернет-аукцион

Чтобы представить реалистичный пример определения агрегатов, в этом разделе мы разберем вымышленное приложение eBidder, реализующее сайт интернет-аукциона, напоминающий eBay. На рис. 19.15 изображены ограниченные контексты, которые были определены в пространстве решений.

Ограниченный контекст «Лот» (Listing) представляет лот, выставленный на продажу, и форму продажи — по фиксированной цене или по цене, сформировавшейся в процессе торгов. Ограниченный контекст «Обсуждения» (Disputes) реализует обсуждение лота между продавцом и участниками торгов. Ограниченный контекст «Членство» (Membership) реализует членство на сайте eBidder. Наконец, ограниченный контекст «Учетная запись продавца» (Selling Account)

реализует возможность оплаты и других действий, связанных с покупкой. В этом разделе мы рассмотрим принципы проектирования агрегатов только на примере ограниченного контекста «Лот».

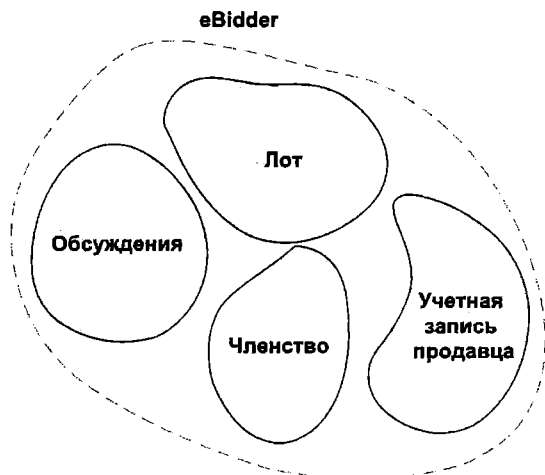


Рис. 19.15. Ограниченные контексты в приложении eBidder

На рис. 19.16 изображена предметная модель ограниченного контекста «Лот». Полный исходный код этого приложения доступен в загружаемом пакете с примерами.

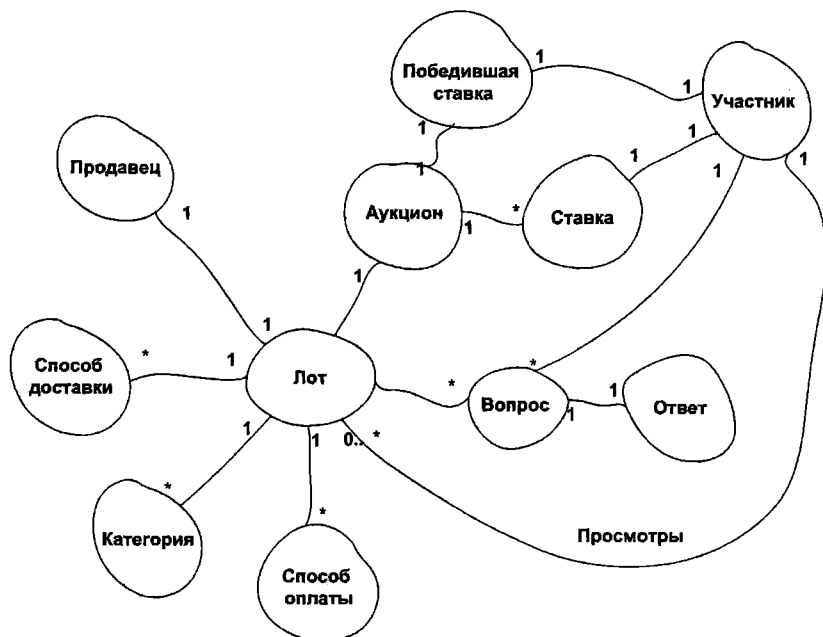


Рис. 19.16. Агрегаты в ограниченном контексте «Лот»

Экземпляр сущности `Listing` представляет лот, выставленный продавцом на продажу. После продажи он может быть доставлен покупателю одним из множества способов, также он может входить в какие-то категории и предусматривать разные способы оплаты. Ограниченный контекст лота может хранить вопросы, возникающие у участников торгов относительно товара; на эти вопросы могут существовать ответы, данные продавцом. Каждому лоту ставится в соответствие форма продажи, например аукционная, когда цена формируется в ходе торгов, в которых могут принимать участие несколько покупателей, делающих ставки. В любой момент, после первой же ставки, на аукционе всегда имеется победившая ставка. Наконец, участник может просматривать лоты, не участвуя в торгах.

Согласование с инвариантами

Основное правило создания агрегатов гласит, что группы предметных объектов должны определяться на основе предметных инвариантов. Об этом уже говорилось ранее в этой главе и подробно будет обсуждаться далее.

Приступая к определению агрегатов, необходимо выявить объекты, работающие вместе и требующие согласования друг с другом для выполнения сценариев использования. Ниже перечислены некоторые предметные правила, формирующие основу для принятия решения о том, какие объекты должны объединяться в агрегаты в ограниченном контексте лота.

- Каждый лот должен принадлежать хотя бы одной категории и определять не менее одного способа оплаты и одного способа доставки.
- Лот предлагается для продажи на аукционе. Аукцион имеет дату начала и окончания торгов и хранит наивысшую ставку.
- Предлагая свою ставку, участник должен указать максимальную сумму, которую он готов заплатить за выбранный лот. Однако текущей победившей ставкой становится сумма, минимально необходимая для победы. Если второй участник сделает ставку, аукцион автоматически поднимет победившую ставку до суммы, необходимой для победы одного из участников. Каждая ставка регистрируется как ставка.
- Участник может задать вопрос о лоте. Продавец может представить ответ.
- В ходе торгов может быть сделано множество ставок, но текущая цена определяется победившей ставкой.
- Участники могут просматривать лоты.

Участники могут задавать вопросы, касающиеся лотов; однако нет никаких инвариантов, которые требовали бы объединения данных из вопросов и лотов, кроме ссылки, которая может быть реализована посредством свойства, хранящего идентификатор. Лот, как понятие, может существовать независимо от вопросов, и вопросы не зависят ни от каких других предметных объектов, кроме ответов. Поэтому границы агрегата можно провести вокруг вопросов и ответов, как показано на рис. 19.17.

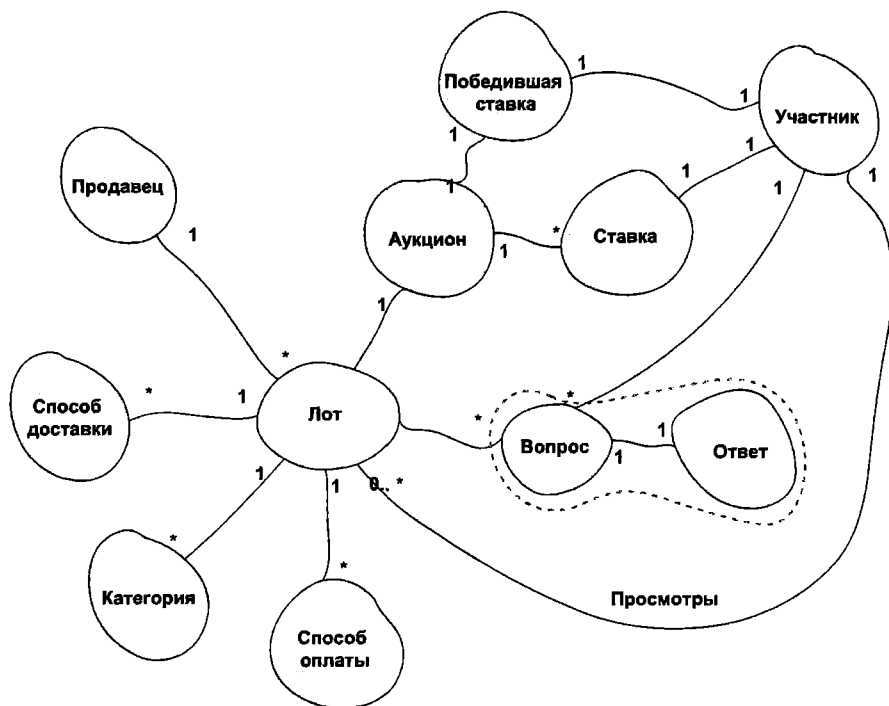


Рис. 19.17. Определение границ агрегата вопроса

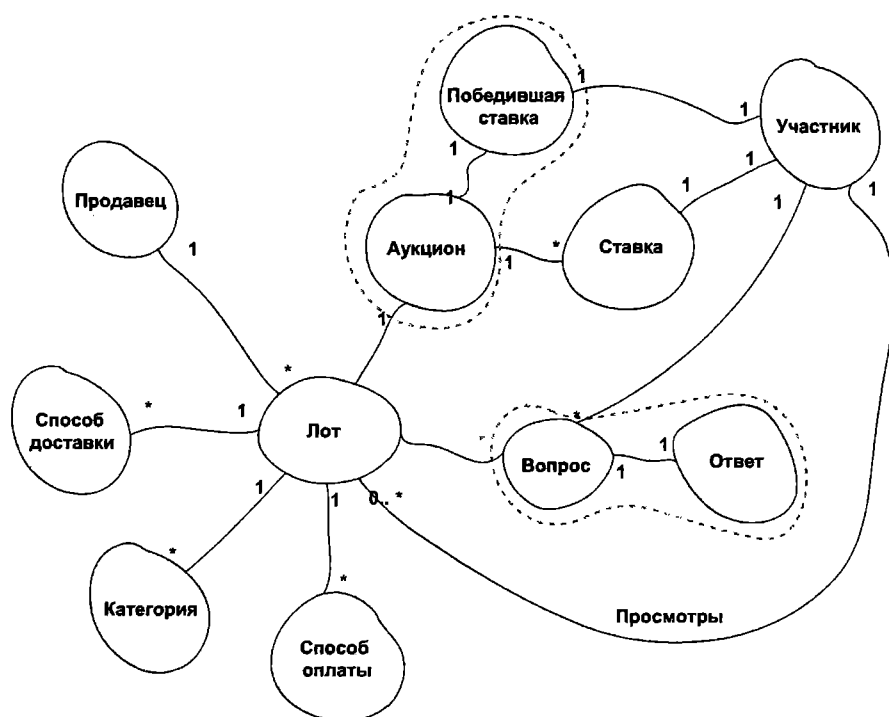


Рис. 19.18. Определение границ агрегата аукциона

Аукцион представлен в формате лота. Он хранит дату начала и окончания торгов, текущую победившую ставку, включая максимальную сумму, до которой участник готов поднять свою ставку. Чтобы ставка была принята, аукцион должен быть активным, а величина победившей ставки должна быть меньше величины ставки, предлагаемой участником. Работа аукциона никак не зависит от особенностей лота. На основе этой информации можно провести еще одну границу агрегата вокруг предметных объектов, представляющих аукцион и победившую ставку, как показано на рис. 19.18.

Лот хранит всю информацию о продаваемом товаре, включая категорию, которой он принадлежит, способы оплаты и способы доставки. Инвариант требует, чтобы лот включал способ доставки, категорию и способ оплаты. Соответственно границы агрегата лота можно определить, как показано на рис. 19.19.

Ставка — это хронологическое событие; она может быть заключена в собственный агрегат, потому что не вовлечена ни в какие инварианты. Участник торгов и продавец совместно используют только идентификаторы, поэтому они тоже могут быть выделены в собственные агрегаты. Участник может просматривать аукцион, но из-за того, что эта операция не имеет никаких инвариантов, объект, представляющий ее, является всего лишь контейнером с идентификатором лота и идентификатором участника; она тоже может быть выделена в отдельный агрегат, как показано на рис. 19.20.

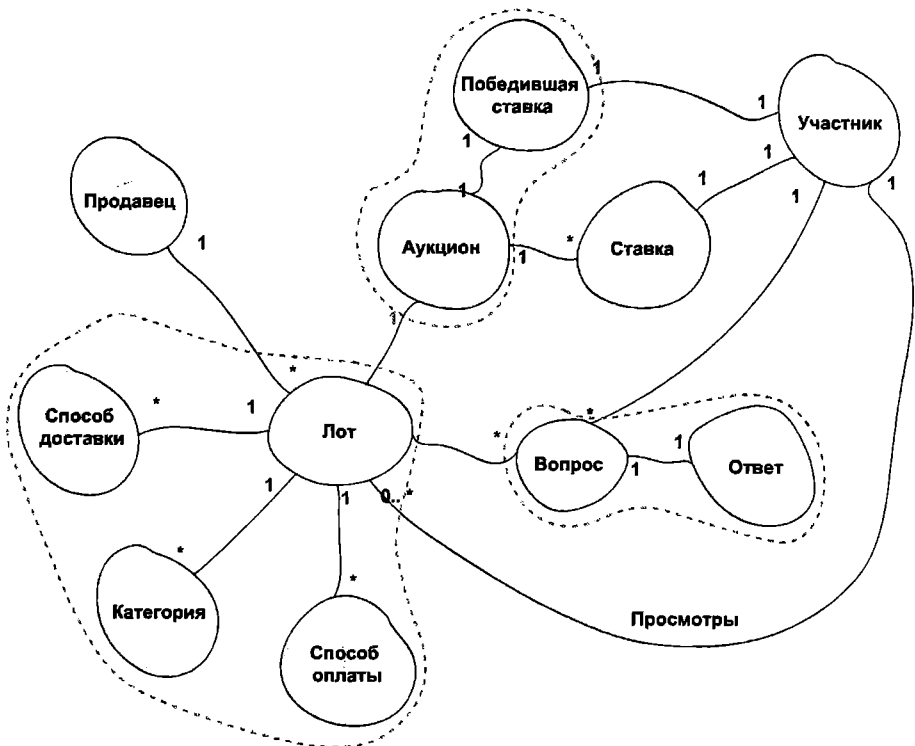


Рис. 19.19. Определение границ агрегата лота

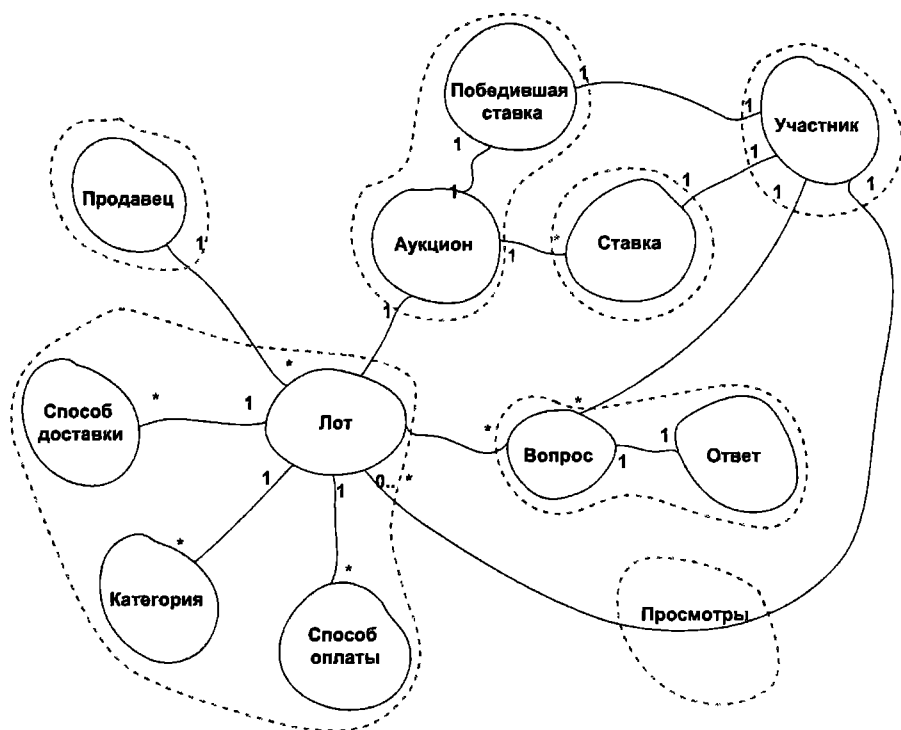


Рис. 19.20. Определение границ всех агрегатов

Согласование с транзакциями

Границы агрегатов желательно согласовать с границами транзакций, потому что чем большее число агрегатов изменяется в одной транзакции, тем выше вероятность отказа операций, выполняющихся параллельно. Поэтому старайтесь ограничить каждый агрегат единственным сценарием использования, чтобы сохранить работоспособность системы. На рис. 19.21 показано, как границы агрегатов аукциона и лота согласуются с транзакциями; каждый агрегат будет изменяться в отдельной транзакции.

Если обнаружится, что в рамках транзакции изменяется более одного агрегата, это может служить признаком недостаточно точного соответствия границ агрегатов и предметной области. Тогда вам придется постараться узнать больше о сценарии использования, обсудив его со специалистами в предметной области или поэкспериментировав со своей моделью. В последнем случае выясните, можно ли решить вопрос, сделав агрегаты потенциально согласованными, чтобы внутри транзакции изменялся только один агрегат. Желательно также узнать, для каких частей системы, по мнению заинтересованных лиц, допустима потенциальная согласованность, и отразить это в структуре агрегатов. В приложении eBidder потенциальная согласованность недопустима для сущностей Auction и WinningBid. Именно поэтому они были объединены в один агрегат.

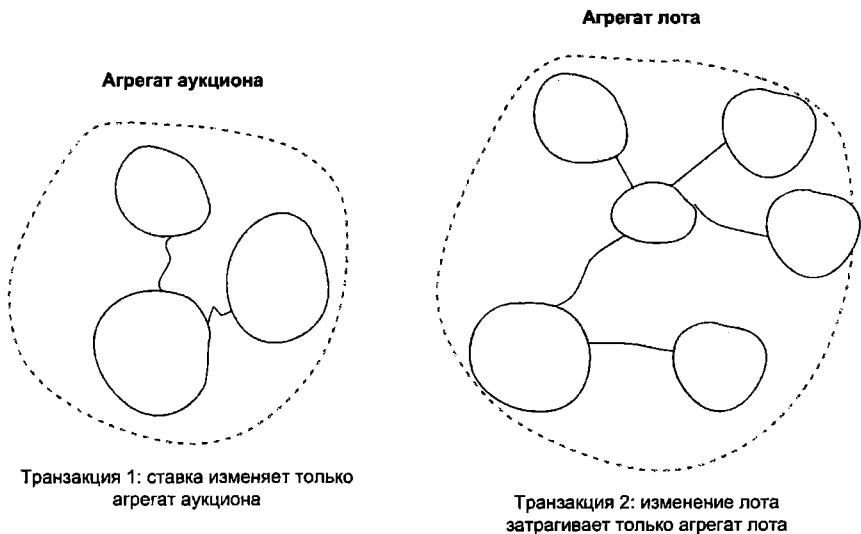


Рис. 19.21. Согласование границ агрегатов с границами транзакций

Игнорируйте требования пользовательского интерфейса

Определение границ агрегатов не должно зависеть от нужд пользовательского интерфейса. На странице с лотом может выводиться информация о продавце, текущая цена на аукционе и описание лота. Если бы границы агрегатов определялись на основе потребностей пользовательского интерфейса, агрегаты получились бы значительно больше, и это могло бы вызывать блокировки, например, если бы продавец вдруг пожелал изменить описание лота в то время, когда участник пытается сделать ставку.

Для удовлетворения нужд пользовательского интерфейса вместо больших агрегатов обычно определяются отображения, включающие несколько агрегатов и содержащие все необходимые данные. Часто такие отображения реализуются в виде прикладных служб, выполняющих множество запросов к репозиторию для загрузки агрегатов и затем отображающих информацию из агрегатов в модель представления.

ПРИМЕЧАНИЕ

Иногда бывает желательно скрыть структуру предметной области, отказавшись от экспортирования внутренних данных агрегатов. Однако вам все еще может потребоваться отображать эту информацию на веб-странице. В главе 26 «Запросы: предметная отчетность» будет показано, как решить эту проблему с применением шаблона проектирования «Посредник» (Mediator), а в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования» будет показано альтернативное решение, основанное на применении шаблона «Хранитель» (Memento).

Несмотря на то что выше рекомендуется заполнять модель представления данными из нескольких агрегатов, этот подход имеет свои недостатки, которые в некоторых случаях могут вынудить вас искать альтернативные решения. Если для заполнения одной страницы требуется выполнить запросы к трем или более репозиториям, это один из явных признаков того, что, возможно, вам стоит поискать другое решение. Обращение к трем базам данных может существенно ухудшить производительность и увеличить нагрузку на серверы. В такой ситуации рассмотрите возможность использования шаблона CQRS, о котором рассказывается в главе 24 «CQRS: архитектура ограниченного контекста».

Избегайте простых коллекций и контейнеров

Часто агрегаты ошибочно представляют в качестве простых коллекций или контейнеров для других объектов. Это может быть опасным заблуждением, ухудшающим ясность предметной модели. Не нужно всякий раз, встретив понятие, похожее на коллекцию или контейнер, слепо полагать, что это агрегат.

Увидев сущность аукциона в приложении eBidder, у многих мог бы появиться соблазн включить в агрегат коллекцию ставок. На первый взгляд такое решение кажется логичным, потому что концептуально аукцион имеет коллекцию ставок. Однако, как было показано в предыдущем разделе, ставки не принадлежат агрегату, потому что в предметной области нет инвариантов, относящихся сразу к обоим понятиям. Такого рода рассуждения могут привести к усложнению иерархий объектов, раздуванию агрегатов и не дать ни одного преимущества от добавления новых агрегатов.

Не закидывайтесь на отношениях владения

Агрегаты не должны повторять модель данных. Связи между объектами не всегда совпадают с отношениями между таблицами в базе данных. Модели данных должны представлять все отношения владения («HAS A» — «имеется») с целью поддержания ссылочной целостности, создания отчетов для бизнес-аналитиков и реализации пользовательских интерфейсов. У лота «имеются» («HAS») вопросы и «имеется» аукцион, но это не означает, что они должны быть объединены в один агрегат. Помните, что агрегат представляет понятие в предметной области, а не является контейнером для объектов. Лот не обязан хранить вопрос или коллекцию вопросов, и нет ни одного предметного инварианта, требующего хранить такую коллекцию. Зачем загружать все вопросы только для того, чтобы добавить новый?

Аукционы и лоты немного отличаются, потому что между ними существует отношение «один к одному». Однако с точки зрения сценариев использования и предметных инвариантов поведение лотов не должно быть совместимо с поведением агрегата аукциона, потому что нет никаких инвариантов, которые объединяли бы их.

Включая предметные объекты в агрегат, не закидывайтесь на отношениях владения; группы должны быть согласованными, и каждый объект не просто должен быть связанным с агрегатом, а обязан определять некоторое его поведение.

Реорганизация агрегатов

Определение границ агрегатов является обратимой и непрекращающейся деятельностью. Совершенно необязательно стремиться создать идеальную структуру с первой попытки. Вместо этого старайтесь постоянно искать более совершенные способы организации агрегатов по мере накопления знаний о предметной области. Поучительным примером может служить добавление в модель новых сценариев использования. Новый сценарий может вовлекать в работу имеющиеся сущности и вскрывать новые отношения. Как следствие, могут возникать новые предметные инварианты, не соответствующие существующей организации агрегатов.

Технические открытия также могут влиять на структуру агрегатов, особенно те из них, которые связаны с улучшением производительности. Если обнаружится, что сохранение — или, что вероятнее, загрузка — агрегата ухудшает производительность до неприемлемого уровня, это может служить сигналом, что агрегат слишком велик. Несомненно, проблемы производительности являются существенной причиной, чтобы переопределить границы агрегатов, даже если концептуально они выглядят идеально. Однако вполне возможно, что проблемы производительности обусловлены неоптимальной организацией агрегатов, поэтому вам может потребоваться посоветоваться со специалистами в предметной области или поэкспериментировать с альтернативными способами организации.

Главное — потребности сценариев использования, а не соответствие реальности

Рассматривайте моделирование агрегатов с точки зрения сценариев использования. Узнайте у специалистов, какие инварианты должны соблюдаться при выполнении сценария. При таком подходе маловероятно попадание в ловушку моделирования реальности. Вместо этого вы будете получать небольшие и функциональные агрегаты.

В листинге 19.5 демонстрируется реализация случая добавления ставки. Обратите внимание, что для добавления ставки не нужна информация о лоте или его категории. Это объясняется тем, что агрегаты лота и аукциона смоделированы отдельно.

Листинг 19.5. Реализация добавления ставки

```
public class BidOnAuctionService
{
    // .....

    public void Bid(Guid auctionId, Guid memberId, decimal amount)
    {
        using (DomainEvents.Register(OutBid()))
        using (DomainEvents.Register(BidPlaced()))
        {
```

```
var member = _memberService.GetMember(memberId);

if (member.CanBid)
{
    var auction = _auctions.FindBy(auctionId);

    var bidAmount = new Money(amount);

    var offer = new Offer(memberId, bidAmount, _clock.Time());

    auction.PlaceBidFor(offer, _clock.Time());
}
}
```

Агрегаты представляют понятия из пространства решений; они не являются отражением реальности. Это всего лишь абстракции, используемые для решения задач наиболее эффективным способом и уменьшения технической сложности. Помня об этом, определяйте границы своих агрегатов с позиции сценариев использования — снаружи внутрь и с учетом предметных инвариантов.

Реализация агрегатов

Организация агрегатов постоянно изменяется под влиянием реализации. Использование хранилищ данных, поддержание согласованности и возможность параллельных операций — вот главные детали реализации, сложные в осуществлении и способные вынудить вас пересматривать границы агрегатов. Понимание допустимости ссылок между агрегатами дается особенно сложно и оно, скорее всего, заставит вас вернуться к процессу проектирования агрегатов. В этой ситуации вам поможет понятие корня агрегата.

Выбор корня агрегата

Чтобы агрегат оставался в согласованном состоянии, его составные части не должны быть доступны повсюду в предметной модели или из уровня служб. Следование этому правилу предотвратит приведение агрегата в несогласованное состояние другими компонентами приложения. Но агрегат должен реализовать некоторое поведение. Для этого в каждом агрегате должна быть выбрана сущность, которая будет играть роль корня агрегата (aggregate root). Все взаимодействия с агрегатом должны выполняться посредством его корня.

Корень агрегата — это сущность, служащая воротами в агрегат. На рис. 19.22 показано, как сущность `Auction` играет роль корня агрегата аукциона.

Корень агрегата координирует все изменения в агрегате и гарантирует, что клиенты не смогут привести его в несогласованное состояние. Он поддерживает все инварианты агрегата, делегируя полномочия другим сущностям и объектам-значениям в агрегате.

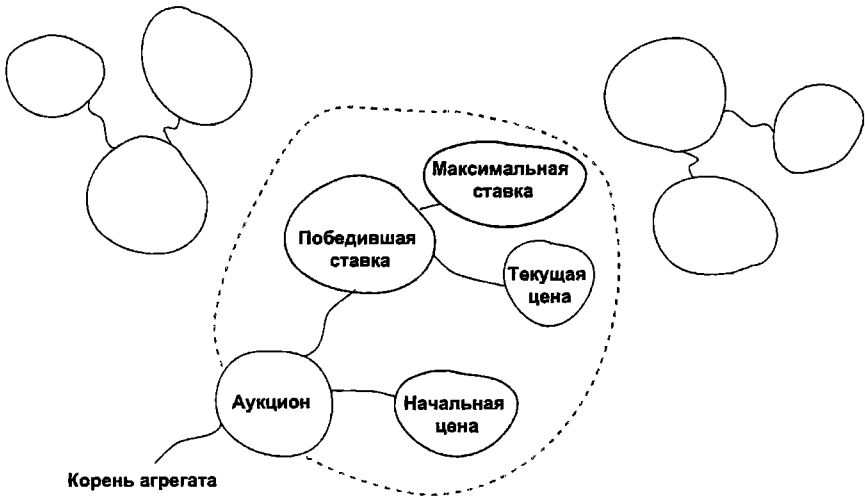


Рис. 19.22. Корни агрегатов — это ворота в агрегаты

Предметные объекты существуют только как части агрегата, как части концептуального целого. Как упоминалось выше, в отсутствие корня клиенты должны были бы иметь доступ к внутренней структуре агрегата и могли бы обходить его функции, взаимодействуя с внутренними сущностями напрямую, как показано в следующем фрагменте:

```
basket.Items.Find(x => x.ProductId == productId).Quantity = newQuantity;
```

Этот пример нарушает инвариант, требующий, чтобы число единиц товара каждого вида не превышало десяти. Переместив это поведение в корень, как показано в следующем фрагменте, можно защитить внутренние объекты агрегата и обеспечить соблюдение инварианта.

```
basket.ChangeQuantityOf(productId, newQuantity);
```

В теле `ChangeQuantity()` находится логика, которая запрещает сущности `Customer` или любому другому клиенту агрегата указывать число единиц товара каждого типа, большее десяти. Таким способом корень выполняет свои обязательства по соответствию инварианту.

Экспортирование поведения

По аналогии с сущностями и другими предметными объектами, весьма желательно экспортировать поведение агрегата так, чтобы модель явно отражала предметные понятия. Для агрегатов это означает необходимость экспортирования выраженных методов в корне, которые могли бы вызывать другие агрегаты. Корень агрегата играет роль посредника для внутренних объектов агрегата и является точкой входа для всех внешних взаимодействий. Эти особенности демонстрирует каркас агрегата аукциона в листинге 19.6.

Листинг 19.6. Сущность Auction

```

public class Auction : Entity<Guid>
{
    public Auction(Guid id, Guid itemId, Money startingPrice, DateTime endsAt)
    {
        ...
    }

    private Guid ItemId { get; set; }
    private DateTime EndsAt { get; set; }
    private Money StartingPrice { get; set; }
    private WinningBid WinningBid { get; set; }
    private bool HasEnded { get; set; }

    public void ReduceTheStartingPrice()
    {
        ...
    }

    public bool CanPlaceBid()
    {
        ...
    }

    public void PlaceBidFor(Offer offer, DateTime currentTime)
    {
        ...
    }
}

```

Здесь методы `ReduceTheStartingPrice()`, `CanPlaceBid()` и `PlaceBidFor()` представляют поведение, экспортируемое корнем агрегата, а значит, и самим агрегатом. Эти методы отражают предметные понятия и выполняют операции с другими членами агрегата, но не открывают к ним доступ. Как обсуждалось выше, это позволяет корню агрегата `Auction` гарантировать, что агрегат всегда будет находиться в согласованном состоянии. Остальные члены агрегата в данном примере определяются ссылками, указывающими на объекты-значения `StartingPrice` и `WinningBid`.

Не все члены агрегата напрямую доступны из корня. Вполне допустимо и иногда даже полезно, чтобы они находились на один-два уровня ниже в дереве объектов. Примерами в агрегате аукциона могут служить объекты-значения `MaximumBid` и `CurrentAuctionPrice`, принадлежащие объекту-значению `WinningBid` и недоступные корню агрегата `Auction`, как показано в листинге 19.7 с каркасом `WinningBid`.

Листинг 19.7. Объект-значение WinningBid

```

public class WinningBid : ValueObject<WinningBid>
{
    public WinningBid(Guid bidder, Money maximumBid,
        Money bid, DateTime timeOfBid)
    {
        ...
    }
}

```

```
    ...
}

public Guid Bidder { get; private set; }
public Money MaximumBid { get; private set; }
public DateTime TimeOfBid { get; private set; }
public Price CurrentAuctionPrice { get; private set; }

public WinningBid RaiseMaximumBidTo(Money newAmount)
{
    ...
}

public bool WasMadeBy(Guid bidder)
{
    ...
}

public bool CanBeExceededBy(Money offer)
{
    ...
}

public bool HasNotReachedMaximumBid()
{
    ...
}
}
```

ВНИМАНИЕ

Не забывайте о предупреждении относительно глубины деревьев объектов, о котором упоминалось в начале главы. Данный пример является скорее исключением, чем правилом.

Защита внутреннего состояния

В предыдущих главах вы узнали, что сокрытие предметной структуры играет важную роль, потому что расширяет возможности реорганизации предметной модели с накоплением знаний о предметной области. Рекомендация с осторожностью подходить к экспортированию методов чтения (getters) и методов записи (setters), о которой говорилось ранее, также применима и к агрегатам. Последовав совету из предыдущего раздела, касающегося экспортирования поведения, вы сделаете большой шаг в направлении защиты внутренней структуры агрегата.

Если еще раз взглянуть на структуру сущности Auction в листинге 19.6, можно заметить, что корень агрегата сосредоточен на экспортировании поведения. Все общедоступные интерфейсы представлены исключительно методами. Все ссылки на внутренние члены объекта определены как приватные переменные-члены. Если агрегат откроет доступ к методам чтения/записи, внутренние элементы агрегата также окажутся открытыми. В этом случае другие предметные объекты или при-

кладные службы смогут обращаться к ним напрямую, что сузит возможности реорганизации предметной модели в будущем. Кроме того, внешние клиенты агрегата смогут ввести его в несогласованное состояние.

Только корни должны иметь глобальную индивидуальность

Возможно, вы слышали из уст практиков DDD упоминание о локальной и глобальной индивидуальности (или идентичности). Эти слова — лишь краткий способ выразить, что корень агрегата имеет глобальную индивидуальность, потому что доступен извне агрегата, а другие члены агрегата имеют локальную индивидуальность, так как они доступны только внутри агрегата. Рисунок 19.23 демонстрирует применение этих двух терминов.

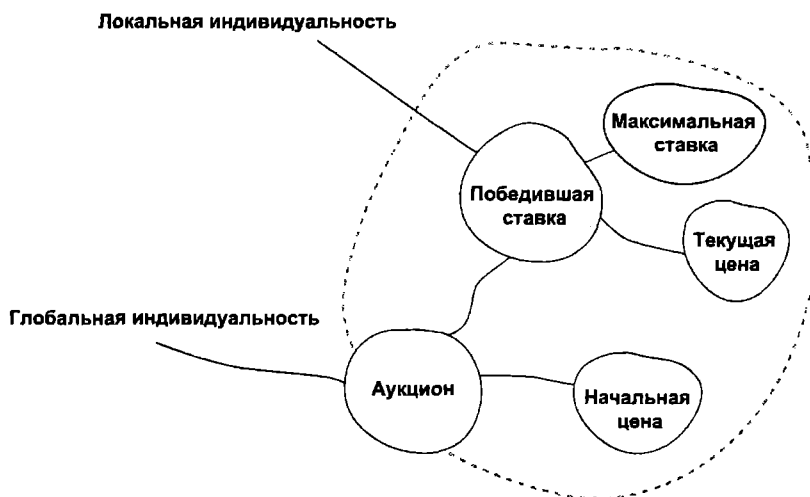


Рис. 19.23. Корни агрегатов имеют глобальную индивидуальность

Ссылка на другие агрегаты

Практически всегда корни агрегатов должны хранить ссылки на идентификаторы других агрегатов, а не на сами объекты. Это важное правило, особенно когда используются инструменты ORM, потому что иначе загрузка одного агрегата из базы данных может повлечь за собой загрузку всех других агрегатов, на которые ссылается данный агрегат. Это может вызвать существенное падение производительности и большие сложности при отладке в том случае, если задействован механизм отложенной загрузки.

В ограниченном контексте лота, в приложении eBidder, агрегат аукциона содержит ссылку на идентификатор лота, выставленного на продажу, как показано в следующем фрагменте. Такое решение объясняется тем, что сущность Listing не является частью агрегата Auction, а включение ссылки на объект может вызвать проблемы с хранением, обсуждавшиеся выше.

```
public class Auction : Entity<Guid>
{
    ...

    private Guid ItemId { get; set; }

    ...
}
```

Как уже было показано в первой части этой главы, ссылка на идентификаторы других корней агрегатов помогает уменьшить накладные расходы при работе с хранилищами и извлекать требуемые агрегаты в службах, только когда это действительно необходимо, как показано в следующем фрагменте:

```
var auction = auctionRepository.FindById(auctionId);
var listing = listingRepository.FindById(auction.ItemId);
// выполнение сценария использования
```

Каждое дополнительное обращение к репозиторию почти наверняка потребует выполнить дополнительный цикл обращения к базе данных. Может показаться, что это неоптимальное решение в сравнении с ситуацией, когда все необходимые данные извлекаются единственным запросом. Следует признать, что сам по себе этот прием не идеален, но в сочетании с планомерным применением шаблона «Агрегат» повсюду в предметной модели он обеспечивает более эффективную стратегию доступа к данным для системы в целом.

Иногда такой подход может приводить к увеличению числа запросов в три и более раза. Если производительность играет важную роль, вы свободно можете принять другое решение. Возможно, создание одного большого агрегата окажется более оптимальным решением. Если остальные агрегаты сохраняют правильную форму, такое решение едва ли повлечет какие-либо проблемы. Однако в таких случаях часто имеет смысл рассмотреть возможность применения шаблона CQRS или регистрации событий.

Важно также отметить, что существует несколько тонкостей относительно допустимых и недопустимых ссылок между агрегатами, которые будут разъясняться на примерах в оставшейся части раздела.

Ничто за пределами агрегата не должно ссылаться на что-либо внутри

Простое правило, неоднократно повторявшееся на протяжении этой главы, гласит: ничто за пределами агрегата не должно ссылаться на его внутренние члены (рис. 19.24). Как уже обсуждалось, это необходимо для защиты внутренней структуры агрегата и предотвращения приведения его в несогласованное состояние.

Клиенты агрегата могут ссылаться только на корень агрегата, за исключением особых случаев, о которых пойдет речь далее.

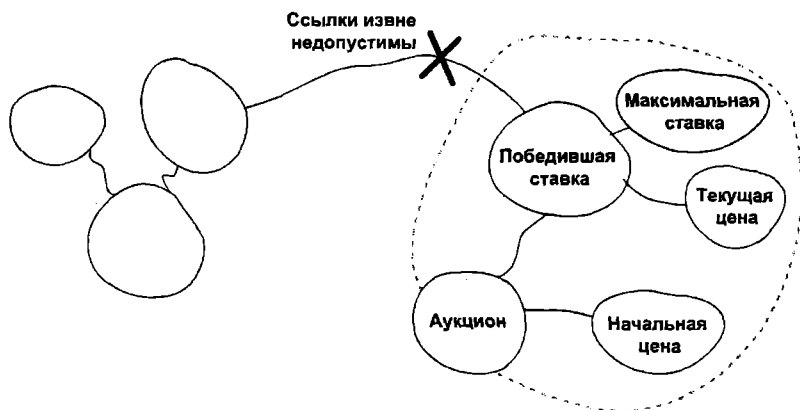


Рис. 19.24. Клиенты агрегата не должны хранить ссылок на его внутренние члены

Корень агрегата может передавать временные ссылки на внутренние объекты

Даже при том, что агрегат не может ссылаться на внутренние члены другого агрегата, вполне допустимо хранить временные ссылки на эти члены, которые, в идеале, должны использоваться в рамках единственного метода.

Временная ссылка на внутренний член агрегата, находящаяся во временной переменной, едва ли может вызвать проблемы с хранилищем. Временная ссылка не является частью дерева объектов агрегата, поэтому ее не требуется загружать при передаче. Но, несмотря на то что в теории все выглядит неплохо, на практике следует проявлять большую осторожность, передавая ссылки на объекты, из-за возможности неправильного их использования и образования зависимостей. Вместо этого предпочтительнее передавать копии объектов или их представления, как показано на рис. 19.25.

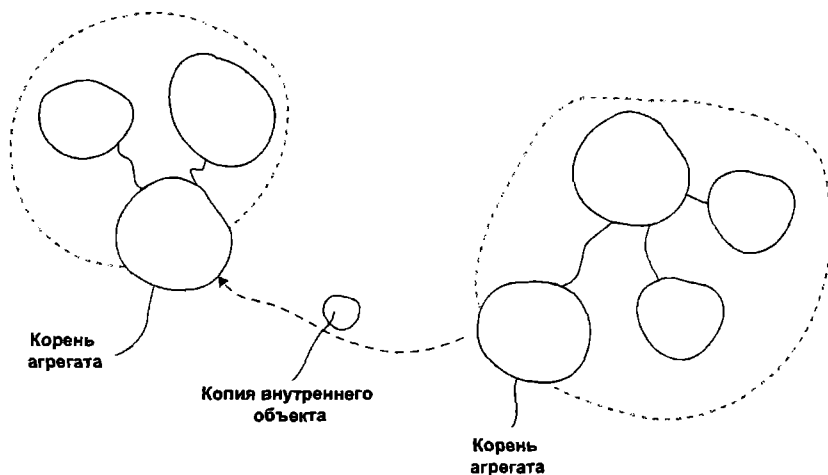


Рис. 19.25. Передача информации между агрегатами посредством копий внутренних объектов

Объекты внутри агрегата могут ссылаться на корни других агрегатов

Как это ни парадоксально, но вполне допустимо, чтобы объекты внутри агрегата ссылались на корни других агрегатов, как показано на рис. 19.26, где объект `WinningBid` внутри агрегата аукциона хранит ссылку на корень другого агрегата. (Это однонаправленное отношение.)

В данном случае под хранением ссылки понимается хранение идентификатора, а не ссылки или указателя на объект, как показано в листинге 19.8. Объект `WinningBid` хранит ссылку (идентификатор) на агрегат `Member`, представляющий участника, разместившего ставку.

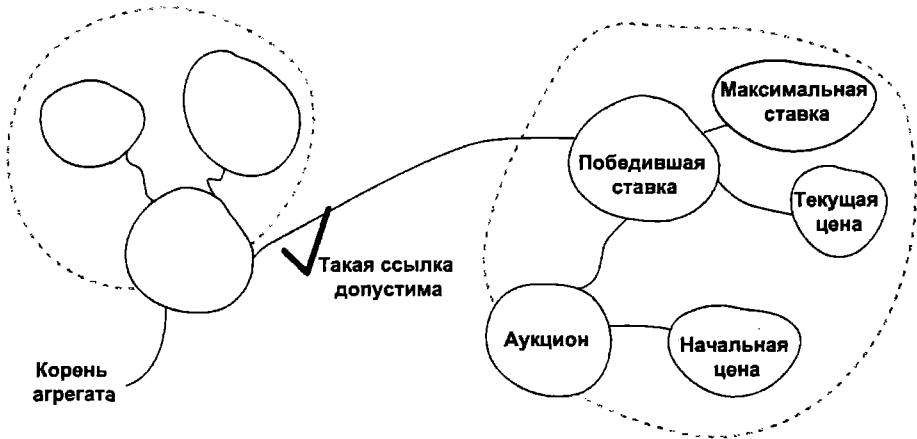


Рис. 19.26. Внутренние объекты могут ссылаться на корни других агрегатов

Листинг 19.8. Объект-значение `WinningBid`

```
public class WinningBid : ValueObject<WinningBid>
{
    public WinningBid(Guid bidder, Money maximumBid,
                     Money bid, DateTime timeOfBid)
    {
        if (bidder == Guid.Empty)
            throw new ArgumentNullException("Bidder cannot be null");

        ...

        Bidder = bidder;

        ...
    }
}
```

```

public Guid Bidder { get; private set; }

...

public bool WasMadeBy(Guid bidder)
{
    return Bidder.Equals(bidder);
}

...

protected override IEnumerable<object>
GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>()
    {
        Bidder, MaximumBid, TimeOfBid, CurrentAuctionPrice
    };
}
}

```

Как обсуждалось выше, агрегаты загружаются из хранилища целиком. Поэтому если объект будет хранить прямую ссылку на другой объект, при загрузке одного агрегата придется также загрузить другой агрегат. Подробнее об этом рассказывается в следующем разделе.

Реализация хранения

С помощью запросов к базе данных можно получить только корни агрегатов. Предметные объекты, являющиеся внутренними компонентами агрегата, доступны только через его корень. Каждому агрегату соответствует репозиторий, представляющий базу данных и позволяющий лишь сохранять и извлекать агрегаты. Очень важно гарантировать соблюдение инвариантов и сохранять агрегаты в согласованном состоянии. Все эти проблемы будут особенно чувствительны, если дочерние объекты агрегата будут напрямую доступны в базе данных.

На рис. 19.27 показано, как следует и как не следует загружать агрегаты из базы данных, если вы хотите гарантировать соблюдение инвариантов и сохранение агрегатов в согласованном состоянии.

Хранение агрегатов целиком упрощает рассуждения о них и их поддержку, так как известно, что существует прямое соответствие между агрегатами и репозиториями. Это также дает возможность размышлять в терминах агрегатов. Всякий раз, когда требуется получить некоторую информацию или воспользоваться некоторым поведением, достаточно знать, какой агрегат загрузить и какой репозиторий использовать. Вышесказанное иллюстрирует листинг 19.9, демонстрирующий тривиальную реализацию прикладной службы `AnswerAQuestionService`, которая использует единственный репозиторий и загружает из него единственный агрегат.

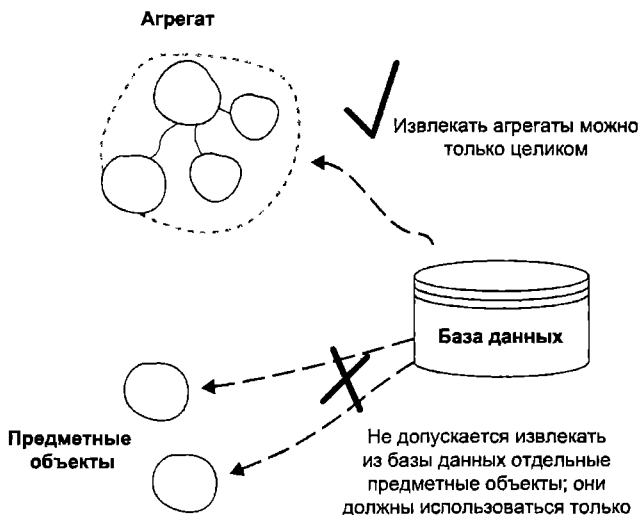


Рис. 19.27. Для защиты согласованности агрегаты должны загружаться из базы данных только целиком

Листинг 19.9. Служба AnswerQuestionService

```
public class AnswerAQuestionService
{
    private IQuestionRepository _questions;

    ...

    public void Answer(Guid questionId, Guid sellerId, string answer,
        bool publishOnListing)
    {
        var question = _questions.FindBy(questionId);

        using (DomainEvents.Register(QuestionAnswered()))
        {
            question.SubmitAnAnswer(
                answer, sellerId, publishOnListing, _clock.Time()
            );
        }
    }
}
```

Определение интерфейса `IQuestionRepository` приводится в следующем фрагменте. Обратите внимание, что определяется лишь две операции — сохранения и загрузки агрегата целиком:

```
public interface IQuestionRepository
{
    Question FindBy(Guid id);

    void Add(Question question);
}
```

Репозитории могут не ограничиваться поиском агрегатов по их идентификаторам. В действительности имеется еще несколько правил, кроме требования сохранять и загружать агрегаты целиком. Например, вам может понадобиться загрузить все вопросы, заданные определенным участником, как показано в следующем фрагменте:

```
public interface IQuestionRepository
{
    Question FindBy(Guid id);

    void Add(Question question);

    IEnumerable<Question> FindByMemberId(Guid memberId);
}
```

Фактические реализации репозитория могут отличаться в зависимости от выбранных технологий хранения данных и доступа к ним. При использовании инструментов ORM, таких как NHibernate, часто достаточно передать корень агрегата в сеанс, чтобы обеспечить сохранение всего агрегата. С другой стороны, при использовании низкоуровневых запросов SQL может понадобиться вручную сохранять каждый член агрегата.

Кому-то может показаться, что в ситуациях, когда требуется единственный предметный объект, загружать агрегат целиком слишком неэффективно или избыточно. Однако, получив некоторый опыт применения DDD, вы убедитесь, что такие сценарии встречаются крайне редко. Возьмем для примера очередь заказов. Возникла ли у вас когда-нибудь потребность загрузить очередь заказа отдельно от остального содержимого агрегата заказа? Что с этим пунктом можно сделать? Сценарии, когда требуется единственный предметный объект, являются исключениями из правил. Чаще они свидетельствуют о неправильно определенных границах агрегатов.

Доступ к предметным объектам для чтения допустим на уровне базы данных

Для выполнения сценария использования часто достаточно извлечь из базы данных единственный агрегат. Однако при составлении отчетов о состоянии предметной области нет необходимости заботиться о соблюдении границ агрегатов. Информация для отчета может выбираться непосредственно на уровне базы данных, без извлечения предметных объектов. Вместо этого можно организовать выполнение моделей представления посредством специализированных запросов. Как показано на рис. 19.28, агрегат должен рассматриваться как предметное понятие, только когда он задействован в сценарии использования, но для составления отчетов вполне можно использовать самые обычные запросы к базе данных.

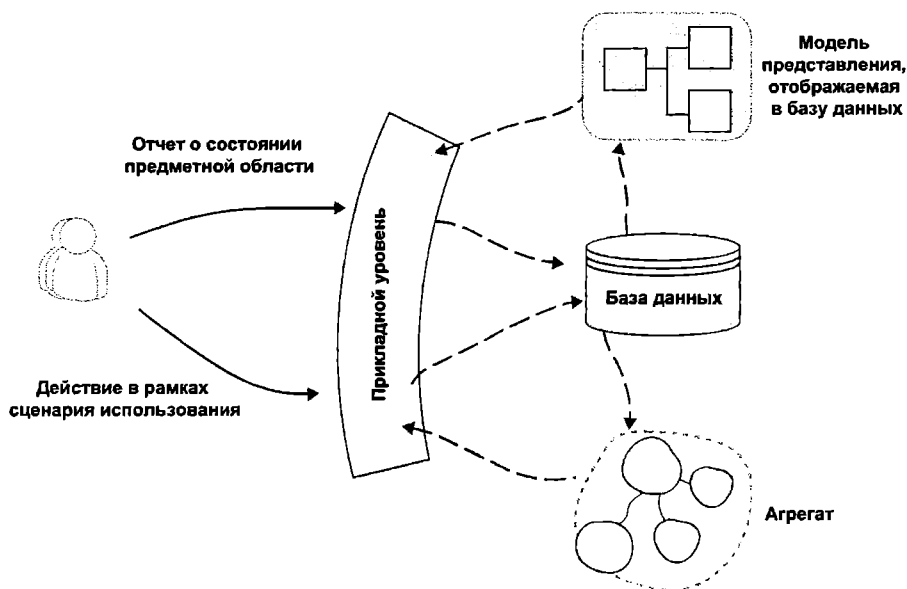


Рис. 19.28. Загрузка агрегатов и прямой доступ к базе данных

ПРИМЕЧАНИЕ

В главе 26 «Запросы: предметная отчетность» приводятся примеры создания отчетов в обход предметной модели.

Операция удаления агрегата должна удалять все, что находится в границах агрегата

При удалении корня агрегата вместе с ним, в рамках той же транзакции, должны удаляться все дочерние предметные объекты. Компоненты, составляющие агрегат, не должны оставаться после удаления корня, потому что без корня они не имеют никакого смысла. Например, бессмысленно хранить очередь заказа после удаления самого заказа.

Избегайте отложенной загрузки

Многие инструменты ORM позволяют организовать загрузку данных, только когда программный код впервые обращается к ним. Несмотря на то что целью отложенной загрузки является увеличение производительности, непредсказуемость этого механизма в действительности может приводить к серьезным проблемам с производительностью и надежностью. Старайтесь использовать отложенную загрузку, только если на то есть чрезвычайно веские причины. Если вам удастся сохранить агрегаты маленькими, отложенная загрузка вам, скорее всего, не понадобится.

Примером ситуации, когда отложенная загрузка может повредить производительности, является пресловутая проблема *n+1 запросов*. Если говорить кратко, эта проблема выражается в извлечении из базы данных каждого элемента некоторой коллекции с помощью отдельного запроса, вместо того чтобы извлечь их все сразу, одним запросом. Для выполнения каждого запроса требуется дополнительное время для установки соединения с базой данных и транспортировки данных по сети, что ухудшает общую эффективность и замедляет доступ к данным. Взгляните, например, на листинг 19.10.

Листинг 19.10. Сущность и корень агрегата Booking

```
// Корень агрегата
public class Booking
{
    ...

    private List<GuestContactDetails> Guests { get; set; }

    public void NotifyGuestsOfConfirmation()
    {
        foreach(var guest in Guests)
        {
            // послать подтверждение по электронной почте
        }
    }

    ...
}
```

Этот фрагмент взят из реализации специализированной предметной модели мероприятий, где группы людей могут бронировать заведения для празднования дней рождений или проведения корпоративных мероприятий. Если бы этот код использовал отложенную загрузку, в каждой итерации цикла `foreach` выполнялся бы дополнительный запрос к базе данных для получения информации о единственном госте. Представьте ситуацию, когда в ресторане бронируются столики на 50 человек для корпоративного празднования Нового года. Программе пришлось бы выполнить 51 запрос к базе данных (один для оформления брони и по одному для каждого гостя). Это не проблема, если вы можете позволить себе падение производительности, но во многих системах ухудшение производительности отрицательно сказывается на восприятии пользователей.

ПРИМЕЧАНИЕ

Дополнительную информацию об отложенной загрузке можно найти на сайте Мартина Фаулера <http://martinfowler.com/eaCatalog/lazyLoad.html>¹. Кроме того, в блоге Айенде Рахин (Ayende Rahien) можно найти интересную статью, связанную с проблемой «n+1 запросов», возникающей при использовании отложенной загрузки: <http://ayende.com/blog/156577/on-umbracos-nhibernates-pullout>.

¹ Вольный перевод на русский: <http://design-pattern.ru/patterns/lazy-load.html>. — Примеч. пер.

Реализация транзакционной согласованности

Как вы уже знаете, для обеспечения согласованности транзакция должна или сохранить агрегат целиком, или отменить все изменения, выполненные в рамках этой транзакции. Число таблиц, в которых хранится агрегат, не имеет значения; когда агрегат сохраняется, все изменения должны быть подтверждены в одной транзакции, чтобы гарантировать, что в случае отказа агрегат не окажется в несогласованном состоянии.

Рисунок 19.29 показывает, что граница агрегата одновременно является границей его согласованности. Изменения во всех членах агрегата должны выполняться или отменяться атомарно.

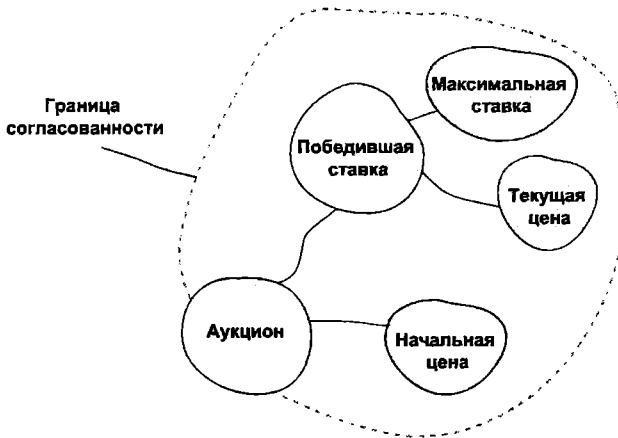


Рис. 19.29. Граница агрегата одновременно является границей согласованности

Под согласованностью понимается, что каждый член агрегата имеет доступ к самому последнему состоянию других членов этого же агрегата. Для агрегата аукциона, например, это значит, что когда бы ни изменился объект `WinningBid`, сущность `Auction` должна иметь доступ к этим изменениям. Это правило легко применяется в границах агрегата, потому что объекты, составляющие агрегат, могут хранить прямые ссылки друг на друга. Следующий фрагмент демонстрирует, как агрегат `Auction` реализует транзакционную согласованность:

```
public class Auction : Entity<Guid>
{
    ...

    private WinningBid WinningBid { get; set; }

    ...
}
```

Когда бы ни изменился объект `WinningBid`, сущность `Auction` всегда будет иметь доступ к измененному значению. Это объясняется наличием в `Auction` прямой

ссылки на объект `WinningBid`. То есть она всегда имеет возможность получить самое последнее значение из самого источника.

За границами агрегата и, соответственно, за границами согласованности правила меняются с точностью до наоборот; согласованность не должна быть настолько строгой. Это объясняется недопустимостью прямого доступа к внутренним объектам агрегата. В примере с агрегатом `Auction` никакие другие агрегаты не должны иметь прямой ссылки на объект `WinningBid`. Поэтому когда происходит изменение `WinningBid`, возникает сложность, связанная с извещением других агрегатов или служб, использующих это значение. Чтобы иметь дело с несвежими данными, они должны быть потенциально согласованными, как было показано ранее в этой главе.

Реализация транзакционной согласованности в значительной степени зависит от используемой технологии хранения и выбора клиентских библиотек. Инструменты ORM, такие как `Hibernate` и `NHibernate`, предлагают свои механизмы транзакций, которые подтверждают или откатывают результаты любых операций, выполнявшихся внутри них. Некоторые базы данных, такие как `RavenDB`, предоставляют похожие интерфейсы.

ПРИМЕЧАНИЕ

Важно помнить, что транзакции и другие технические детали должны быть реализованы в уровне прикладных служб. Дополнительные советы и примеры вы найдете в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования».

Реализация потенциальной согласованности

По существу, потенциальная согласованность реализуется агрегатами как передача копий своих данных другим агрегатам, и, как следствие, агрегаты должны мириться с тем фактом, что полученная ими информация может оказаться устаревшей. Это объясняется тем, что когда один агрегат изменяется, другие агрегаты, получившие частичную копию его состояния, не обновляют эту копию немедленно новыми значениями. Как рассказывалось ранее в этой главе, именно поэтому граница согласованности совпадает с границей агрегата.

Существуют определенные стратегии реализации потенциальной согласованности. Самая простая заключается в том, чтобы выполнять транзакции синхронно. Преимущество этого подхода заключается в минимальном периоде времени, когда агрегаты оказываются несогласованными между собой. Однако это решение не рекомендуется, потому что реализовать его правильно очень сложно. На практике для реализации потенциальной согласованности обычно используется асинхронный подход с использованием внешних технологий, таких как фреймворк `NServiceBus`, о котором рассказывалось в главе 12 «Интеграция посредством обмена сообщениями». Агрегаты в этом случае остаются в несогласованном состоянии дольше, но сама реализация более надежна.

Правила, распространяющиеся на несколько агрегатов

Как вы узнали из этой главы, потенциальная согласованность обычно возникает там, где предметные правила распространяются на несколько агрегатов. Следующий

пример покажет, как реализовать потенциальную согласованность для понятия лояльности в программе электронной коммерции; клиенты, совершившие в течение года покупок более чем на \$1000, получают 10%-ную скидку на все последующие покупки. В первый момент может появиться соблазн организовать изменение заказа и оценки лояльности в одной транзакции. Однако, как вы уже знаете, это увеличивает шанс конфликтов одновременного доступа и ухудшает масштабируемость.

Часто лояльность не тот сценарий, что требует немедленной согласованности. Как клиенту, вам было бы приятно немедленно получить 10%-ную скидку, но, скорее всего, вам придется подождать следующего рабочего дня, чтобы скидка начала действовать. Не забудьте получить одобрение от заинтересованных лиц перед реализацией потенциальной согласованности. Следующий пример демонстрирует реализацию потенциальной согласованности с использованием асинхронных событий в предположении, что заинтересованные лица согласились с правилом вступления в силу скидки на следующий день.

Асинхронная потенциальная согласованность

Асинхронная потенциальная согласованность позволяет получить более надежную систему из-за возможности повторения операций в случае неудачи. Однако для реализации и поддержки асинхронного поведения требуется вводить в действие новые технологии и службы. Если у вас уже есть опыт использования технологий, таких как Kafka, Akka или NServiceBus, для организации обмена сообщениями между ограниченными контекстами, тогда для вас стоимость реализации асинхронной потенциальной согласованности будет не очень высока.

Теоретически согласованность между агрегатами можно реализовать с применением приемов, что использовались в главах с 11-й по 13-ю для реализации потенциальной согласованности между ограниченными контекстами, — с применением асинхронных предметных событий. Этот подход был проиллюстрирован выше на рис. 19.14, где изменение агрегата и публикация события осуществлялись в рамках одной транзакции. Затем, в следующей транзакции, событие обрабатывалось и выполнялись необходимые операции.

Для реализации потенциальной согласованности между агрегатами с применением предметных событий хорошо подойдет шаблон «Предметные события» и его класс `DomainEvents`, инициирующий обработку. Публикация событий внутри обработчика осуществляется с помощью фреймворка сообщений. В листинге 19.11 представлена измененная версия `OrderApplicationService`, использующая интерфейс `IBus` из фреймворка NServiceBus для публикации события размещения заказа.

Листинг 19.11. Метод `PlaceOrder`, отправляющий предметное событие

```
public class OrderApplicationService
{
    ...

    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);
```

```
var order = customer.AddOrder(products);

// публикация выполняется в рамках транзакции
_bus.Publish(new OrderCreated(order));

// подтвердить транзакцию
_unitOfWork.SaveChanges();
}
}
```

ПРИМЕЧАНИЕ

Глава 18 «События предметной области» целиком посвящена предметным событиям и содержит разнообразные примеры.

В некоторый момент в будущем, возможно, в другом потоке выполнения или даже на другом сервере, фреймворк сообщений вызовет обработчик, который запустит вторую транзакцию для обновления оценки лояльности. При использовании NServiceBus обработчик можно было бы реализовать, как показано в листинге 19.12.

Листинг 19.12. Обработка события создания заказа

```
// Обработчик NServiceBus
public class PlaceOrderHandler : IHandleMessages<OrderCreated>
{
    ...

    public void Handle(OrderCreated @event)
    {
        var loyalty = _loyaltyRepository.FindByCustomerId(
            @event.Order.CustomerId
        );

        loyalty.RegisterPurchaseAmount(@event.Order.TotalSpend);

        // опубликовать следующее событие
    }
}
```

ВНИМАНИЕ

Если сохранение в базе данных и публикация события выполняются в рамках одной транзакции, необходимо узнать, как используемая система обмена сообщениями поддерживает двухфазное подтверждение (two-phase commit, 2pc). В случае с NServiceBus и MS SQL Server подумайте о возможности использования координатора распределенных транзакций (Microsoft Distributed Transactions Coordinator, MSDTC), как описывается в документации к NServiceBus (<http://docs.particular.net/nservicebus/transactions-message-processing>).

Использование фреймворков обмена сообщениями не единственный способ реализации асинхронного решения. Другой вариант, описывавшийся в главе 11 «Введение в интеграцию ограниченных контекстов», — интеграция посредством базы

данных. В данном примере вместо отправки события устанавливается флаг в базе данных. Через определенные интервалы времени может запускаться некоторое задание, проверяющее появление новых заказов и изменяющее оценку лояльности клиентов, как показано на рис. 19.30.

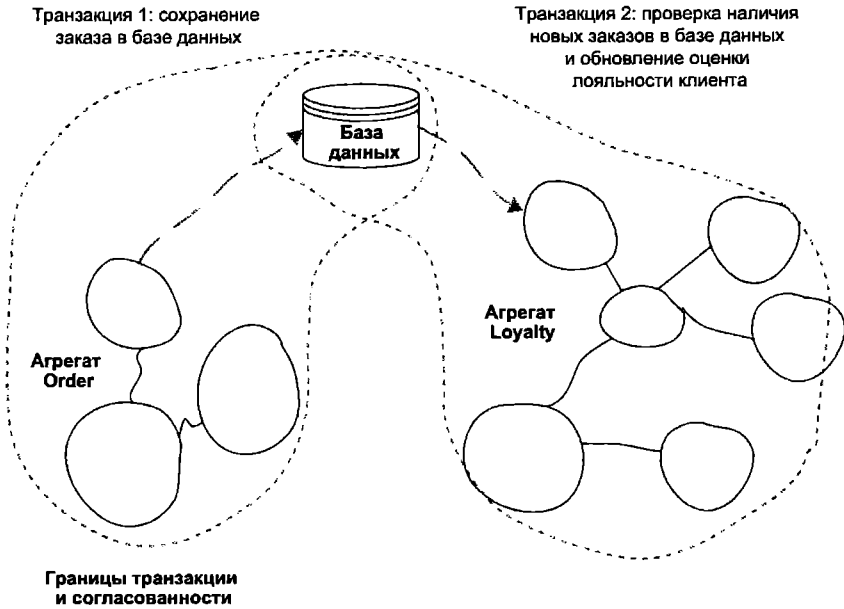


Рис. 19.30. Потенциальная согласованность с использованием базы данных

Очевидно, что реализация асинхронной потенциальной согласованности сопряжена с множеством сложностей, среди которых применение технологий обмена сообщениями, организация асинхронной обработки и периодическая проверка. Поэтому рекомендуется в каждом конкретном случае вновь оценивать, какой вариант подойдет лучше: обновление нескольких агрегатов в одной транзакции или применение одного из подходов реализации потенциальной согласованности.

Поддержка одновременного доступа

Корень агрегата может защитить доступ к внутренним компонентам. Но он не может предотвратить приведение агрегата в несогласованное состояние, если сразу несколько пользователей попытаются изменить его. В многопользовательских окружениях, где есть вероятность, что сразу несколько пользователей могут изменить состояние предметного объекта и сохранить его в базе данных, должен иметься механизм, гарантирующий, что изменения, внесенные одним пользователем, не окажут отрицательного влияния на состояние параллельно выполняющихся транзакций, запущенных другими пользователями.

Существует две формы управления одновременным доступом: оптимистическая и пессимистическая. Оптимистическая форма управления предполагает отсут-

ствие проблем при одновременном изменении состояния предметных объектов несколькими пользователями и известна как «побеждает последний». Для некоторых систем это вполне разумное поведение; однако когда требуется, чтобы объекты находились в согласованном состоянии при извлечении из базы данных, необходимо использовать пессимистическую форму управления.

Пессимистическая форма управления одновременным доступом имеет множество разновидностей — от блокировки таблицы данных на время извлечения записи до сохранения копии содержимого предметного объекта и сравнения ее с версией в хранилище перед сохранением. Такая проверка помогает убедиться, что сущность не изменялась с момента ее извлечения. Поддержку одновременного доступа можно реализовать добавлением в агрегаты версии или отметки времени. В этом разделе, для проверки изменения предметной сущности с момента извлечения ее из базы данных, используется номер версии. Перед подтверждением транзакции после изменения сущности номер ее версии сравнивается с номером версии в базе данных. Если номера версий не совпадают, возбуждается исключение. Это поможет гарантировать неизменность сущности после ее извлечения.

Для реализации поддержки одновременного доступа используется типичный трехступенчатый подход, на первом этапе которого в корень добавляется номер версии:

```
public class Loyalty : Entity<Guid>
{

    public int Version { get; private set; }

}
```

Затем, как показано в листинге 19.13, перед подтверждением транзакции номер версии увеличивается. Но перед этим выполняется сравнение номера версии в сущности с последним номером версии в базе данных.

Листинг 19.13. Реализация поддержки параллельного доступа в репозитории

```
public class LoyaltyRepository : ILoyaltyRepository
{
    ...

    public void Save(Loyalty loyalty)
    {
        var currentVersion = GetCurrentVersionOf(loyalty);

        if (currentVersion != loyalty.Version)
        {
            var error = string.Format(
                "Expected: {0}. Found: {1}",
                Loyalty.Version, currentVersion
            );
        }
    }
}
```

```
        throw new OptimisticConcurrencyException(error);
    }

    // ошибок нет - продолжить сохранение
}
```

Следует признать, что данный подход к поддержке одновременного доступа является всего лишь стратегией ослабления риска. В зависимости от особенностей библиотеки, используемой для доступа к данным, все еще могут оставаться короткие интервалы времени между проверкой номера версии и успешным подтверждением транзакции. К сожалению, в этот короткий промежуток агрегат может сохранить другая транзакция. Именно поэтому так желательно перенести управление параллельным доступом на уровень хранилища данных или библиотеки доступа к нему. Многие библиотеки, такие как NHibernate, Entity Framework и клиент RavenDB, предусматривают возможность настройки управления параллельным доступом.

ПРИМЕЧАНИЕ

Похожие примеры добавления поддержки параллельного доступа к сущностям с регистрацией событий можно найти в главе 22 «Регистрация событий». В тех примерах показаны подходы к реализации проверки согласованности и на уровне приложения, и на уровне хранилища данных.

Ключевые идеи

- Уменьшение числа двунаправленных связей между предметными объектами в модели уменьшает сложность реализации и выявляет направленность связей, что помогает понять, где разместить предметную логику.
- Ограничивайте связи, добавляя критерии, чтобы уменьшить техническую сложность решения.
- Используйте ссылки на объекты для организации связей, только если это необходимо для поддержки инварианта; в противном случае ссылайтесь на объекты посредством идентификаторов.
- Включайте в модель только связи, необходимые для поддержки предметных инвариантов; не нужно слепо моделировать реальность или организацию данных в коде.
- Агрегаты помогают разбить большое дерево объектов на маленькие группы, чтобы уменьшить сложность технической реализации предметной модели.
- Агрегаты должны представлять предметные понятия, а не абстрактные коллекции предметных объектов.
- Согласовывайте границы агрегатов с предметными инвариантами.

- Агрегаты определяют границы согласованности и гарантируют безопасность состояния предметной модели.
- Агрегаты обеспечивают проведение границ транзакций на правильном уровне детализации и помогают избежать блокировок на уровне базы данных.
- Старайтесь создавать маленькие агрегаты, чтобы уменьшить вероятность блокировки транзакций и сложность поддержки согласованности.
- Корень агрегата — это сущность, входящая в состав агрегата и играющая роль ворот, ведущих в агрегат. Корень агрегата гарантирует согласованность состояния агрегата и соблюдение инвариантов.
- Корень агрегата имеет глобальную индивидуальность; предметные объекты внутри агрегата могут иметь индивидуальность только в контексте корня этого агрегата.
- Предметные объекты за пределами агрегата могут ссылаться только на корень агрегата.
- Когда удаляется корень агрегата, вместе с ним должны удаляться все предметные объекты, входящие в агрегат.
- Никакой предметный объект за границами агрегата не может ссылаться на внутренние объекты этого агрегата.
- Корень агрегата может возвращать копии или ссылки на внутренние предметные объекты для использования в предметных операциях, но любые изменения этих объектов должны выполняться только через корень агрегата.
- Агрегаты, а не отдельные предметные объекты должны сохраняться в базе данных и извлекаться из нее посредством репозитория.
- Внутренние объекты агрегата могут хранить ссылки на корни других агрегатов.
- В идеале границы транзакций должны совпадать с границами агрегатов.
- Между агрегатами может возникать несогласованность; используйте предметные события для изменения агрегатов в отдельных транзакциях.

20

Фабрики

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Как фабрики помогают отделить использование объектов от их создания
- Применение фабричных методов в агрегатах
- Использование фабричных методов для восстановления
- Когда использовать фабрики

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке **Download Code** (Загружаемый код). Примеры кода для главы 20 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Как реализовать создание, сохранение и извлечение предметных объектов так, чтобы не обременять предметную модель техническими деталями? Жизненный цикл сложных предметных объектов может потребовать координации их создания и сохранения. Поддержку инвариантов, соответствующих созданию, можно реализовать с помощью шаблона «Фабрика» (Factory), предложенного «Бандой четырех» (Gang of Four)¹.

¹ «Банда четырех» (англ. Gang of Four, сокращенно GoF) — распространенное название группы авторов (Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson), Джон Влиссидес (John Vlissides)), выпустивших в 1995 году известную книгу «Design Patterns» о шаблонах проектирования; переводное издание: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. — Примеч. пер.

Роль фабрики

В предметных моделях, представляющих обширные предметные области, агрегаты, сущности и объекты-значения могут получаться очень сложными. С появлением необходимости знать все тонкости создания допустимых экземпляров зависимых объектов может ухудшиться выразительность модели. Необходимость знания инвариантов других объектов нарушает принцип единственной ответственности (Single Responsibility Principle, SRP). В этой связи рекомендуется отделять использование от конструирования и явно инкапсулировать логику создания в фабричный объект, если она слишком сложная или если такое решение может улучшить выразительность. Создание объектов не является предметной задачей, но выполняется внутри предметного уровня приложения. Вам редко придется говорить о фабриках со специалистами в предметной области, но, несмотря на это, фабрики играют важную роль.

Отделение использования от создания

Фабрики можно использовать для восстановления предметных объектов из хранилища или для создания новых предметных объектов, заключив в них сложную логику создания. Шаблон «Фабричный метод» (factory method) принадлежит группе шаблонов проектирования для создания объектов, разработанных «Бандой четырех», и предназначен для решения проблемы создания объектов без указания класса создаваемого объекта.

Главной целью шаблона «Фабрика» является сокрытие сложностей, связанных с созданием объектов. К таким сложностям относятся: какому классу создаваемого объекта отдать предпочтение, если клиент опирается на некоторую абстракцию, или как осуществлять поддержку инвариантов. Второй целью фабрики является ясное выражение намерений разных вариантов создания экземпляров объектов, чего обычно не удастся достичь с использованием одних только конструкторов. Стандартная реализация фабричного класса должна содержать статический метод, возвращающий абстрактный класс или интерфейс. Часто, но не всегда, клиент передает некоторую информацию, используя которую фабрика определяет, экземпляр какого класса требуется создать и вернуть. Возможность абстрагировать ответственность за создание подклассов позволяет клиентскому коду полностью игнорировать особенности создания зависимых объектов. Это соответствует принципу инверсии зависимости (Dependency Inversion Principle). Еще одно преимущество шаблона «Фабрика» заключается в централизации логики создания объектов; если позднее потребуется изменить порядок создания объектов, это можно будет легко осуществить, отыскав и изменив логику создания, не затронув код, использующий ее.

Соккрытие внутренних деталей

Добавляя новые элементы в агрегат, важно не раскрыть структуру этого агрегата. В листинге 20.1 демонстрируется прикладная служба, которая должна знать все тонкости корзины, чтобы добавить в нее новый товар.

Листинг 20.1. Код прикладной службы, требующий знания всех тонкостей создания допустимых объектов

```
namespace Application
{
    public class AddProductToBasket
    {
        // .....

        public void Add(Product product, Guid basketId)
        {
            var basket = _basketRepository.FindBy(basketId);

            var rate = TaxRateService.ObtainTaxRateFor(product.Id, country.Id);

            var item = new BasketItem(rate, product.Id, product.price);

            basket.Add(item);

            // ...
        }
    }
}
```

Метод прикладной службы в листинге 20.1 должен знать все особенности логики конструирования `BasketItem`. Однако он не должен отвечать за создание новых объектов, так как его главная задача — координация.

Во избежание утечки деталей из агрегата можно добавить в него фабричный метод. В листинге 20.2 показано определение объекта `Basket` с новым методом `Add`, который скрывает от прикладной службы реализацию сохранения элементов в корзине. Как видите, ответственность за создание была передана агрегату `Basket`, способному гарантировать согласованность своих внутренних коллекций благодаря поддержке инвариантов. Клиент — прикладная служба — в результате получился намного проще, и ему не требуется знать, как `Basket` хранит свои элементы.

Листинг 20.2. Смещение ответственности в предметный уровень

```
namespace Application
{
    public class AddProductToBasket
    {
        // .....

        public void Add(Product product, Guid basketId)
        {
            var basket = _basketRepository.FindBy(basketId);
            basket.Add(product);

            // ...
        }
    }
}
```

```

    }
}

namespace DomainModel
{
    public class Basket
    {
        // .....

        public void Add(Product product)
        {
            if (Contains(product))
                GetItemFor(product).IncreaseItemQuantitBy(1);
            else
            {
                var rate = TaxRateService.ObtainTaxRateFor(product.Id,
                                                            country.Id);

                var item = new BasketItem(rate, product.Id, product.price);

                _items.Add(item);
            }
        }
    }
}

```

Однако в таком случае возникает другая проблема: **Basket** теперь отвечает за создание зависимости для **BasketItem**, которая не является ее собственностью (объект **rate**, представляющий налоговую ставку). Чтобы создать допустимый элемент на основе некоторого продукта, сущность **Basket** должна передать налоговую ставку. Чтобы создать объект, представляющий эту налоговую ставку, требуется вызвать службу **TaxRateService**. Теперь сущность **Basket** взвалила на себя вторичную ответственность и должна знать, как создать допустимый элемент и где получить нужные налоговые ставки.

Чтобы избавить **Basket** от дополнительной ответственности и скрыть внутреннюю организацию **BasketItem**, можно ввести фабричный объект, заключающий логику создания **BasketItem**, в том числе и получение налоговой ставки. В листинге 20.3 показано, как делегировать создание элемента фабричному объекту **BasketItemFactory**. Если впоследствии порядок вычисления налогов изменится или для создания **BasketItem** потребуются получить еще какую-то информацию, это никак не затронет класс **Basket**.

Листинг 20.3. Делегирование создания объекта фабрике

```

namespace DomainModel
{
    public class Basket
    {
        // .....

        public void Add(Product product)

```



```

    {
        if (Contains(product))
            GetItemFor(product).IncreaseItemQuantitBy(1);
        else
            _items.Add(BasketItemFactory.CreateItemFor(product,
                deliveryAddress));
    }
}

public class BasketItemFactory
{
    public static void CreateBasketFrom(Product product, Country country)
    {
        var rate = TaxRateService.ObtainTaxRateFor(product.Id, country.Id);

        return new BasketItem(rate, product.Id, product.price);
    }
}
}

```

Вызов службы, возвращающей ставку налога, можно было бы перенести в конструктор `BasketItem`, но такое решение нельзя назвать удачным. Конструкторы должны оставаться максимально простыми. Если обнаружится, что для создания объекта требуется сложная логика, это может служить признаком необходимости создать фабрику.

Соккрытие решения о выборе типа создаваемого объекта

В предметном уровне использование фабрики также является допустимым в целях абстрагирования от класса создаваемого объекта, когда имеется несколько вариантов и выбор из них не является ответственностью клиентского класса. Клиентский код может использовать интерфейс или абстрактный класс и оставлять ответственность за выбор конкретного типа фабричному классу.

В листинге 20.4 показано, что сущность заказа может создавать партии для отгрузки с помощью отдельного фабричного метода. Интересно отметить, что в этом методе для создания действительной партии необходимо выбрать транспортную компанию. Класс `Order` не может знать, как выбрать тип транспортной компании, поэтому данную операцию он делегирует классу `CourierFactory` и использует в своей работе абстрактный класс `Courier`, а фабрика `CourierFactory` создает и возвращает конкретную реализацию.

Листинг 20.4. Выбор правильного подкласса с помощью фабрики

```

namespace DomainModel
{
    public class Order
    {
        // ...

        public Consignment CreateFor(IEnumerable<Item> items, destination)
    }
}

```

```

    {
        var courier = CourierFactory.GetCourierFor(items,
                                                    destination.Country);

        var consignment = new Consignment(items, destination, courier);

        SetAsDispatched(items, consignment);
        return consignment;
    }
}

public class CourierFactory
{
    public static Courier GetCourierFor(IEnumerable<Item> consignmentItems,
                                        DeliveryAddress destination)
    {
        if (AirMail.CanDeliver(consignmentItems, destination))
        {
            return new AirMail(consignmentItems, destination);
        }
        else if (TrackedService.CanDeliver(consignmentItems, destination))
        {
            return new TrackedService(consignmentItems, destination);
        }
        else
        {
            return new StandardMail(consignmentItems, destination);
        }
    }
}
}

```

Фабричный класс сам обращается к методу `CanDeliver()` разных реализаций `Courier`, чтобы найти ту, которая сможет обработать заданную партию. Конструкторы каждой из реализаций абстрактного класса `Courier` также проверяют возможность доставки, используя тот же метод внутренне.

Фабричные методы в агрегатах

Фабрики не всегда должны быть реализованы как статические классы. Агрегаты также могут включать фабричные методы для сокрытия от клиентов сложностей, связанных с созданием объектов. В листинге 20.5 демонстрируется класс `Basket`, уже обсуждавшийся выше, который теперь экспортирует возможность создания `WishListItem` из `BasketItem`. Полученный объект `WishListItem` — это сущность для агрегата `WishList`. Сущность `Basket` ничего не делает с объектом `WishListItem` после его создания, но она содержит все необходимые исходные данные. Фабричный метод извлекает клиентов и прикладные службы от необходимости знать, как извлекать объекты `BasketItem` и преобразовывать их в объекты `WishListItem`, переместив логику в сущность `Basket`, где она выглядит более естественной.

Листинг 20.5. Фабричный метод в агрегате

```
namespace DomainModel
{
    public class Basket
    {
        // .....

        public WishListItem MoveToWishList(Product product)
        {
            if (BasketContainsAnItemFor(product_snapshot))
            {
                var wishListItem = WishListItemFactory.CreateFrom(
                    GetItemFor(product));

                RemoveItemFor(product);

                return wishListItem;
            }
        }
    }
}
```

Фабричные методы могут также создавать целые агрегаты. В листинге 20.6 показано, как с помощью `Account` создать заказ, если на счете имеется достаточная для этого сумма.

Листинг 20.6. Фабричный метод, конструирующий агрегат

```
namespace DomainModel
{
    public class Account
    {
        // .....

        public Order CreateOrder()
        {
            if (HasEnoughCreditToOrder())
                return new Order(this.Id,
                                this.PaymentMethod, this.Address);
            else
                throw new InsufficientCreditToCreateAnOrder();
        }
    }
}
```

Можно определить еще один фабричный метод, который будет вызывать другой конструктор объекта `Order`, если требуется пропустить проверку суммы на счете. Как можно заметить в листинге 20.7, без использования фабричных методов было бы сложно понять назначение разных конструкторов объекта `Order`.

Листинг 20.7. Альтернативный фабричный метод в агрегате

```
namespace DomainModel
{
    public class Account
    {
        // .....

        public Order CreateAnOrderIgnoringCreditRating()
        {
            return new Order(this.Id, this.PaymentMethod,
                             this.Address, PaymentType.PayBeforeShipping);
        }
    }
}
```

Фабрики для восстановления

Если вы не используете инструменты объектно-реляционного отображения, способные отображать модель данных в предметную модель со всеми необходимыми преобразованиями, или если вы извлекаете предметные объекты из некоторого файла или из унаследованной системы посредством веб-службы, вам определенно понадобится гарантировать соответствие всем инвариантам при восстановлении предметных объектов. Использование фабрик для восстановления предметных объектов лишь немногим сложнее, чем создание объектов.

В листинге 20.8 показан репозиторий, который извлекает из внешней службы корзину в виде простой модели хранения данных, а затем обращается к объекту `BasketFactory`, находящемуся в предметном уровне, для создания экземпляра `Basket` на основе полученных данных. Здесь также можно видеть, что `BasketFactory` дополнительно создает объект `DeliveryOption` на основе исходных данных и затем с его помощью проверяет, можно ли использовать его для данного экземпляра корзины. Если полученный экземпляр `DeliveryOption` окажется недопустимым, он не используется; вместо него используется шаблон проектирования «Нейтральный объект» (Null Object), обеспечивающий выбор нового объекта `DeliveryOption`.

Листинг 20.8. Использование фабрики для восстановления объекта

```
namespace Infrastructure
{
    public class BasketRepository : IBasketRepository
    {
        // .....

        public Basket FindBy(Guid id)
        {
            BasketDTO rawData = ExternalService.ObtainBasket(id.ToString());
            var basket = BasketFactory.ReconstituteBasketFrom(rawData);

            return basket;
        }
    }
}
```

```
    }  
  }  
}  
  
namespace DomainModel  
{  
    public class BasketFactory  
    {  
        // ...  
  
        public static Basket ReconstituteBasketFrom(BasketDTO rawData)  
        {  
            Basket basket;  
  
            // ...  
  
            var deliveryOption = new DeliveryFactory.Create(  
                rawData.DeliveryOptionId);  
  
            if (deliveryOption.CanBeUsedFor(rawData.Items))  
                basket.Set(deliveryOption);  
            else  
                basket.Set(DeliveryFactory.CreateNonChosen());  
  
            // .....  
  
            return basket;  
        }  
    }  
}
```

Гарантировать соблюдение всех инвариантов при восстановлении агрегата, безусловно, важно; однако так же важно учитывать реалии ситуации, в которой происходит восстановление хранимого объекта. В предыдущем сценарии можно было бы возбудить исключение и инициировать некоторую процедуру исправления объекта, но в данном случае проще создать агрегат и заменить параметры доставки (delivery option) пустым объектом, чтобы позднее пользователь смог выбрать соответствующий вариант доставки.

Практическое использование фабрик

Фабрики могут оказаться эффективным средством увеличения ясности предметной модели и сохранения ее выразительности. Однако их следует использовать только там, где их применение действительно дает ожидаемый эффект, а не повсюду, где требуется создавать экземпляры объектов. Используйте фабрики, если они оказываются выразительнее конструкторов или дают дополнительные удобства там, где имеется более одного конструктора. Используйте фабрики там, где элементы логики конструирования не являются ответственностью зависимого класса.

Ключевые идеи

openhide.biz

- Фабрики отделяют использование объектов от их создания.
- С помощью фабричных классов можно скрыть детали создания сложных предметных объектов от клиентского кода и других предметных объектов.
- Фабрики можно использовать для создания соответствующих экземпляров исходя из потребностей вызывающего кода.
- Фабричные методы могут предотвратить необходимость экспортировать внутреннее состояние агрегата.
- Когда имеется множество конструкторов для разных целей, фабричные объекты могут помочь улучшить выразительность кода.

21

Репозитории

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Роль и ответственность шаблона «Репозиторий»
- Ошибочные представления о шаблоне «Репозиторий»
- Отличия между предметной моделью и моделью хранения
- Стратегии сохранения предметной модели, основанные на возможностях фреймворка сохранения

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке **Download Code** (Загружаемый код). Примеры кода для главы 21 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Как реализовать создание, сохранение и извлечение предметных объектов так, чтобы не обременять предметную модель техническими деталями? Для предметных объектов, которые должны сохраняться и извлекаться позднее, необходимо предусмотреть возможность их записи в постоянное хранилище. Чтобы избежать размывания границ ответственности между предметной логикой и инфраструктурным кодом, можно использовать репозитории. Репозитории гарантируют простоту сохранения предметных агрегатов, скрывая за фасадом коллекций истинные инфраструктурные механизмы. Репозитории помогут устранить лишние технические сложности из предметной модели.

Репозитории

Репозитории используются для управления сохранением агрегатов и их извлечением, гарантируя при этом отделение предметной модели от модели данных. Они являются промежуточным звеном между этими двумя моделями и прячут за

фасадом коллекций сложности взаимодействий с платформой хранения данных и любыми фреймворками сохранения. Репозитории реализуют те же возможности, что коллекции в .NET, необходимые для сохранения и удаления агрегатов, и дополнительно могут включать средства для явного выполнения запросов и получения суммарной информации о коллекции агрегатов.

Репозитории имеют три основных отличия от традиционных стратегий доступа к данным:

- Ограничивают доступ к предметным объектам, позволяя извлекать и сохранять только корни агрегатов, благодаря чему все изменения и инварианты будут обрабатываться только самими агрегатами.
- Поддерживают обобщенный интерфейс, скрывающий технологии, которые действительно используются для сохранения и извлечения агрегатов.
- Самое главное: они определяют границы между предметными моделями и моделями данных.

Так на что же похож репозиторий? В листинге 21.1 показан интерфейс для репозитория, который управляет сохранением и извлечением агрегатов, представляющих клиентов. Интерфейс определяется в пространстве имен предметной модели, потому что является ее частью, а реализация находится в пространстве имен технической инфраструктуры.

Листинг 21.1. Интерфейс репозитория для хранения информации о клиентах

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        void Add(Customer customer);
        void Remove(Customer customer);
    }
}
```

Типичным клиентом репозитория является уровень прикладных служб. Репозиторий определяет все методы доступа к данным, которые могут потребоваться приложению для выполнения бизнес-задачи. Реализация находится в пространстве имен инфраструктуры и для выполнения основной работы обычно опирается на фреймворк сохранения данных. В листинге 21.2 представлена реализация интерфейса `ICustomerRepository` с применением фреймворка `NHibernate`.

Листинг 21.2. Реализация `ICustomerRepository`

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
        private ISession _session;

        public CustomerRepository (ISession session)
```



```
{
    _session = session;
}

public IEnumerable<Customer> FindBy(Guid id)
{
    return _session.Load<Order>(id);
}

public void Add(Customer customer)
{
    _session.Save(customer);
}

public void Remove(Customer customer)
{
    _session.Delete(customer);
}
}
```

Реализация репозитория в данном случае просто вызывает методы интерфейса `ISession`, реализованного во фреймворке `NHibernate`, и действует как шлюз к модели данных. Фактическое отображение предметных объектов в модель данных осуществляется посредством XML, дополнительного кода или отображения атрибутов. Так как `NHibernate` поддерживает сохранение объектов `POCO` (Plain Old C# Object – старый добрый объект C#, используемый, когда не требуется, чтобы предметные объекты наследовали или реализовали какие-либо классы), он способен отобразить предметную модель непосредственно в модель данных. Не волнуйтесь, если у вас нет опыта использования `NHibernate` или инструментов объектно-реляционного отображения (Object Relational Mappers, ORM); некоторые примеры реализаций в конце главы покажут вам, как использовать `RavenDB`, `Entity Framework` и `NHibernate` для реализации репозитория.

Главное преимущество репозитория заключается в том, что они позволяют вынести технические детали из предметной модели и сосредоточиться исключительно на предметных понятиях и логике.

Ошибочные представления о шаблоне

Существует масса неверных и ошибочных представлений о шаблоне «Репозиторий», среди которых чаще всего встречаются мнения, что это избыточная формальность и ненужная абстракция. И действительно, с простыми предметными моделями шаблон «Репозиторий» оказывается чересчур сложным и вместо него предпочтительнее использовать простой доступ к данным или, что еще лучше, напрямую обращаться к фреймворку сохранения данных. Однако когда моделируется решение для сложной предметной области, шаблон «Репозиторий» дополняет модель. Он помогает вскрыть намерения, стоящие за операцией извлечения

агрегата, и может быть реализован так, чтобы придать операциям смысл в терминах предметной области, а не технического фреймворка.

«Репозиторий» — это антишаблон?

Многие разработчики негативно отзываются о шаблоне «Репозиторий», называя его антишаблоном за то, что он скрывает возможности лежащего в основе фреймворка сохранения данных. В действительности же это преимущество шаблона. Вместо открытого интерфейса к модели данных, поддерживающего любые запросы и операции изменения, шаблон «Репозиторий» делает операции извлечения более четкими за счет использования именованных методов запроса и ограничения доступа уровнем агрегата. Более четкие представления об операциях извлечения упрощают настройку запросов и, что еще более важно, позволяют выразить цели запроса не на языке SQL, а в терминах, понятных специалистам в предметной области. Помимо запросов, шаблон «Репозиторий» предполагает экспортирование методов доступа к хранилищу с осмысленными именами вместо типовых методов создания, чтения, изменения и удаления (create, read, update и delete — CRUD), имена которых почти ничего не говорят об их предназначении.

Достоинство шаблона «Репозиторий» состоит не в простоте тестирования программного кода и не в простоте замены одного механизма доступа к данным другим. Он помогает отделить предметную модель от технического фреймворка поддержки хранения данных, чтобы модель могла развиваться без влияния со стороны используемой технологии. Без промежуточного репозитория технические детали использования инфраструктуры хранения данных почти наверняка проникли бы в предметную модель и ослабили ее целостность и, в конечном счете, практичность.

Негативное отношение к шаблону «Репозиторий» проистекает, похоже, от непонимания, где и зачем он должен использоваться. Этот шаблон может пригодиться лишь при определенных условиях. Точно так же слепое применение шаблона «Предметная модель» или любого другого шаблона проектирования может привести к усложнению решения вместо его упрощения.

Отличия между предметной моделью и моделью хранения

Если роль хранилища играет реляционная база данных и для доступа к ней используется инструмент ORM, избавляющий от необходимости вручную отображать объекты в строки и свойства в столбцы и писать запросы на языке SQL, — у вас может возникнуть вопрос: зачем обременять себя реализацией шаблона «Репозиторий», если уже есть фреймворк, помогающий абстрагироваться от технологии хранения данных? Дело в том, что инструмент ORM абстрагирует только реляционную модель данных. Он всего лишь помогает представить модель данных в объектно-ориентированной манере для упрощения работы с данными в программном коде.

Модель хранения, в которую ORM отображает ваши данные, отличается от предметной модели, как показано на рис. 21.1. Модель хранения представляет собой

организацию реляционной базы данных; она состоит из таблиц и столбцов, а не из сущностей и объектов-значений. Иногда предметные модели и модели данных могут выглядеть очень похожими и даже полностью совпадать друг с другом, но концептуально они имеют существенные отличия. Предметная модель — это абстракция предметной области со своим поведением и языком. Модель данных — это лишь структура хранилища, предназначенного для хранения состояния предметной модели в определенный момент времени. Инструменты ORM выполняют отображение в модель данных и упрощают ее использование. Они имеют мало общего с предметной моделью. Роль же репозитория состоит в том, чтобы разделить две модели и не позволить им проникнуть друг в друга. Инструмент ORM — это не репозиторий, но репозитории можно использовать для сохранения состояний предметных объектов.

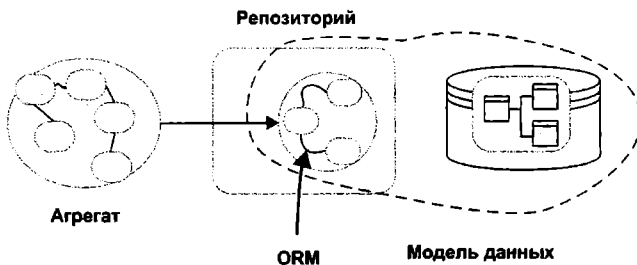


Рис. 21.1. Инструмент ORM используется для отображения между предметной моделью и моделью хранения

Если предметная модель схожа с моделью данных, развитые инструменты ORM, такие как NHibernate, смогут отобразить предметные объекты непосредственно в хранилище. В противном случае лучше использовать полностью отдельную модель данных, нежели ухудшать выразительность предметной модели. Далее в этой главе исследуются некоторые возможности достижения этой цели. Хранилища документов не имеют подобной проблемы и позволяют сохранять предметные модели безо всяких компромиссов.

Преимущество наличия модели данных, отдельной от предметной модели, состоит в возможности развивать предметную модель без необходимости постоянно думать о хранении данных и о том, как они будут представлены в хранилище. Конечно, данные так или иначе придется сохранять, и вам определенно потребуются пойти на какие-то компромиссы, но только когда эти компромиссы действительно необходимы и только после того, как предметная модель будет готова, а не загодя.

Обобщенный репозиторий

Разработчики обожают многократно использовать один и тот же программный код. Они обобщают понятия, создают базовые классы, которые затем можно использовать с разными вариациями, даже при том, что отдельные решения могли бы лучше выразить соответствующие им понятия. Вам часто придется видеть обобщенный контракт репозитория, определяемый в предметной модели. В лис-

тинге 21.3 показан один из таких интерфейсов со всеми необходимыми операциями, включая открытый для расширения метод запросов.

Листинг 21.3. Обобщенный интерфейс репозитория

```
namespace DomainModel
{
    public interface IRepository<TAggregate, TId>
    {
        IEnumerable<TAggregate> FindAllMatching(Expression<Func<T, bool>> query);
        IEnumerable<TAggregate> FindAllMatching(string query);
        TAggregate FindBy(TId id);
        void Add(TAggregate aggregate);
        void Remove(TAggregate aggregate);
    }
}
```

Контракт репозитория, объявленный в предметной модели, мог бы наследовать этот интерфейс и напоминать, например, содержимое листинга 21.4.

Листинг 21.4. Наследование обобщенного интерфейса репозитория

```
namespace DomainModel
{
    public interface ICustomerRepository : IRepository<Customer, Guid>
    {
    }
}
```

Проблема такого контракта заключается в его предположении, что все агрегаты поддерживают одно и то же поведение и имеют одни и те же потребности. Однако некоторые агрегаты, например, могут быть доступны только для чтения и не поддерживать метод удаления. Когда агрегат не поддерживает некоторое понятие, в реализации репозитории часто приходится возбуждать исключение, как показано в листинге 21.5.

Листинг 21.5. Конкретная реализация репозитория клиентов

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
        // ....

        public void Remove(Customer aggregate)
        {
            throw new NotImplementedException(
                "A customer cannot be removed from the collection.");
        }
    }
}
```

Другая проблема интерфейсов — утечка абстракции методов `FindAllMatching`. Предоставляя метод, открытый для расширения, вы лишаетесь возможности управлять запросами и оптимизировать стратегии извлечения. Такая поддержка специализированных запросов может быстро привести к серьезным проблемам производительности в реляционных базах данных.

Попытка использовать обобщенную стратегию во всех репозиториях не самое удачное решение. Обобщенная стратегия ничего не говорит о намерениях, стоящих за операциями извлечения агрегатов. Как в любом другом случае, желательно четко обозначать свои цели. Определяйте свои репозитории, исходя из их конкретных потребностей, и четко обозначайте свои цели, выбирая осмысленные имена для методов, как показано в листинге 21.6.

Листинг 21.6. Специализированный интерфейс репозитория клиентов

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        IEnumerable<Customer> FindAllThatAreDeactivated();
        void Add(Customer customer);
    }
}
```

Когда дело дойдет до реализации, в ней найдется место для обобщенного репозитория. Обобщенный репозиторий, стоящий за конкретной реализацией, можно использовать, чтобы избежать дублирования кода. Как показано в листинге 21.7, такой подход позволяет добавить в конкретный контракт выгоды от повторного использования кода.

Листинг 21.7. Репозиторий клиентов использует прием композиции и не наследует обобщенный репозиторий

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
        private IRepository<Customer, Guid> _customersRepository;

        public Customers(IRepository<Customer, Guid> customersRepository)
        {
            _customersRepository = customersRepository;
        }

        // ....

        public IEnumerable<Customer> FindAllThatAreDeactivated()
        {
            _customersRepository.FindAllMatching(new
                CustomerDeactivatedSpecification());
        }
    }
}
```

```
    }

    public void Add(Customer customer)
    {
        _customersRepository.Add(customer);
    }
}
}
```

При использовании фреймворка сохранения данных вам не требуется абстрагироваться от него; его можно использовать непосредственно в конкретной реализации репозитория. В листинге 21.8 показан пример использования фреймворка RavenDB.

Листинг 21.8. Репозиторий клиентов, не абстрагирующий от использования RavenDB

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
        private IDocumentSession _documentSession;

        public Customers(IDocumentSession documentSession)
        {
            _documentSession = documentSession;
        }

        // ....

        public IEnumerable<Customer> FindAllThatAreDeactivated()
        {
            return _documentSession.Query<Customer>()
                                   .Where(x => x.Deactivated == true)
                                   .ToList();
        }

        public void Add(Customer customer)
        {
            _documentSession.Store(customer);
        }
    }
}
```

Здесь фреймворк сохранения данных используется непосредственно, потому что абстрагирование от него не дает никаких выгод. Помните, что главная цель шаблона «Репозиторий» — убрать технические проблемы из предметной модели. Все остальные части приложения не требуется абстрагировать и скрывать детали реализации, потому что такой подход будет только вводить в заблуждение тех, кому придется читать код.

Стратегии хранения агрегатов

Стратегия проектирования определяет форму агрегатов, а окружение влияет на выбор способов хранения предметных объектов. Следует, однако, помнить, что первичными являются все-таки предметные объекты — они должны создаваться без оглядки на требования, предъявляемые к их хранению; эту проблему должен решать репозиторий. Предметные объекты должны быть свободны от любого инфраструктурного кода и по возможности реализованы как РОСО. Компромиссы допустимы, только когда не остается иного выхода.

Использование фреймворка сохранения, способного отобразить предметную модель в модель данных без компромиссов

При отображении предметных объектов в реляционную базу данных в абсолютно новом окружении и использовании инструмента ORM, поддерживающего сохранение любых предметных объектов, есть возможность отобразить предметную модель непосредственно в модель данных, как показано на рис. 21.2.

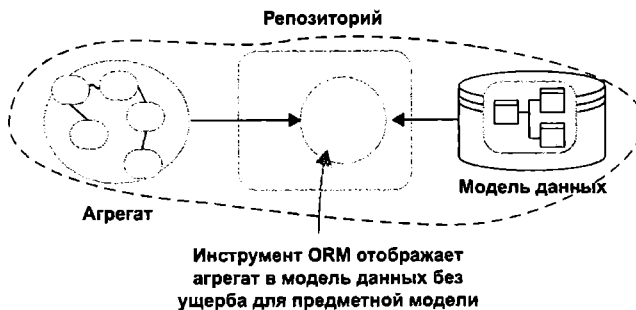


Рис. 21.2. Инструмент ORM отображает предметную модель в модель данных

Обычно для сохранения предметных объектов и поддержки хранения полностью инкапсулированной предметной модели с приватными методами чтения и записи инструменты ORM используют механизмы отражения (или рефлексии). Некоторые фреймворки ORM требуют внести небольшие дополнения в предметные объекты, например конструкторы без параметров в случае с NHibernate, но это не слишком высокая цена за «бесшовное» отображение между предметной моделью и моделью данных.

При использовании документоориентированного хранилища, когда фреймворк может просто сериализовать агрегаты, как показано на рис. 21.3, вы получаете возможность отображать агрегаты непосредственно, без дополнительного преобразования в модель данных.

Фреймворк, избавляющий предметную модель от решения инфраструктурных задач, позволит развивать модель без компромиссов и может оказать существенное влияние на подходы к моделированию.

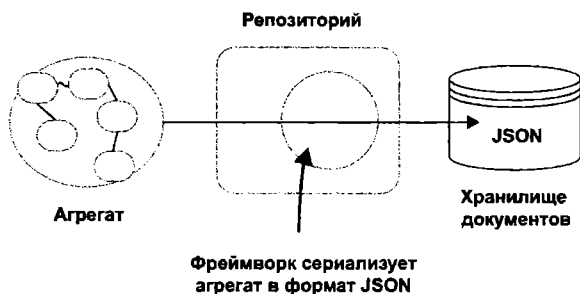


Рис. 21.3. Агрегат можно сериализовать и сохранить

Использование фреймворка сохранения, не способного отобразить предметную модель без компромиссов

При использовании фреймворка, не обеспечивающего независимость предметной модели от особенностей хранилища, следует избрать иной подход к сохранению и извлечению предметных объектов, чтобы они могли оставаться свободными от инфраструктурных проблем. Существует множество способов достижения этого, но все они оказывают влияние на формирование предметной модели и агрегатов. Это — компромисс, на который придется пойти, чтобы получить работоспособное приложение.

Общедоступные методы чтения и записи

Самый простой способ поддержать возможность хранения предметных объектов — снабдить все их свойства общедоступными методами чтения и записи. Проблема такого решения в том, что оно оставляет предметные объекты открытыми, и это может привести к нарушению их состояния клиентами, которые смогли обойти методы изменения состояния, предусмотренные вами. Избежать этого можно с помощью методик обзора и управления кодом. Однако экспортирование свойств действительно упростит сохранение предметной модели, потому что репозиторий получит простой доступ к состояниям предметных объектов.

В листинге 21.9 приводится предметный объект `Basket` с общедоступными свойствами, хранящими коллекцию элементов и стоимость их доставки. Если клиент напрямую добавит новый элемент в коллекцию, минуя метод `Add`, стоимость доставки может остаться прежней.

Листинг 21.9. Все свойства в классе `Basket` объявлены общедоступными

```
namespace DomainModel
{
    public class Basket
    {
        public Money DeliveryCost { get; set; }
        public IList<Item> Items { get; set; }

        public void Add(Product product)
```



```

    {
        if (AlreadyContains(product))
            FindItemFor(product).IncreaseItemQuantityBy(1);
        else
            Items.Add(BasketItemFactory.CreateItemFor(product));

        DeliveryCosts = DeliveryOption.CalculateDeliveryCostsFor(Items);
    }
}

```

Но с другой, положительной, стороны, реализация репозитория может хранить модель данных отдельно от предметной модели. В листинге 21.10 показано, как это может выглядеть.

Листинг 21.10. Репозиторий корзины скрывает сложности отображения объекта `Basket` в хранилище данных

```

namespace Infrastructure.Persistence
{
    public class BasketRepository : IBasketRepository
    {
        // ....

        public Basket FindBy(Guid id)
        {
            Basket basket;

            var basketDataModel = FindDataModelBy(id);

            if (basketDataModel != null)
            {
                basket = new Basket();
                basket.DeliveryCost = basketDataModel.DeliveryOptionCost;

                foreach (var item in basketDataModel.Items)
                {
                    basket.Items.Add(ItemFactory.CreateFrom(item));
                }

                return basket;
            }
        }
    }
}

```

В листинге 21.10 можно видеть, что после извлечения модель данных отображается в новый экземпляр предметного объекта, представляющего корзину. Это позволяет сделать предметную модель независимой от любых особенностей хранилища, но за это приходится платить общедоступностью свойств предметной модели.

Шаблон «Хранитель»

Если нежелательно экспортировать свойства предметной модели и требуется сохранить ее полностью закрытой, можно воспользоваться шаблоном «Хранитель» (memento). Шаблон, предложенный «Бандой четырех», позволяет восстановить предыдущее состояние объекта. Предыдущее состояние экземпляра может храниться в базе данных. Для этого агрегат должен уметь производить снимок собственного состояния, пригодный для сохранения. Также агрегат должен знать, как воссоздать себя из того же снимка состояния. Важно понимать, что снимок — это не модель данных, а всего лишь состояние агрегата, не зависящее от особенностей используемого фреймворка сохранения. Репозиторий в этом случае все еще должен отобразить снимок в модель данных. Рисунок 21.4 представляет этот шаблон в графическом виде.

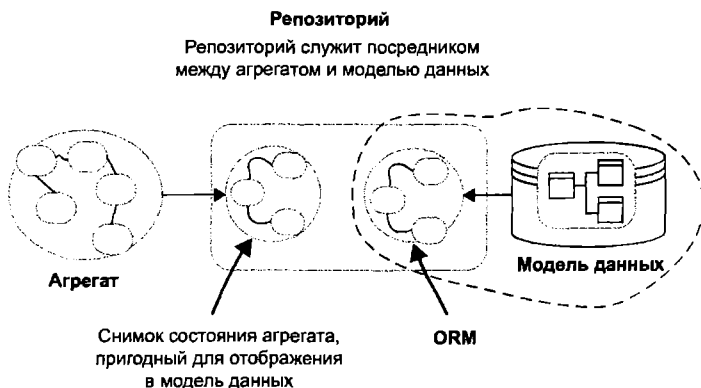


Рис. 21.4. Шаблон «Хранитель» позволяет отобразить снимок предметной модели в модель хранения

В листинге 21.11 представлена реализация шаблона «Хранитель» в агрегате `Basket`.

Листинг 21.11. Реализация шаблона «Хранитель»

```
namespace DomainModel
{
    public class Basket
    {
        private Basket(BasketSnapshot snapshot)
        {
            Items = CreateBasketItemsFrom(snapshot.ItemsSnapshot);
            DeliveryCost = snapshot.DeliveryCost;
        }

        // ...

        private Money DeliveryCost { get; set; }
```

```
private IList<Item> Items { get; set; }

public IEnumerable<ItemView> ShowItems()
{
    return Items.ConvertToItemView();
}

public void Add(Product product)
{
    if (AlreadyContains(product))
        FindItemFor(product).IncreaseItemQuantityBy(1);
    else
        Items.Add(BasketItemFactory.CreateItemFor(product));

    DeliveryCosts = DeliveryOption.CalculateDeliveryCostsFor(Items);
}

public BasketSnapshot GetSnapshot()
{
    var snapshot = new BasketSnapshot();

    snapshot.ItemsSnapshot = CreateItemSnapshotFrom(Items);
    snapshot.DeliveryCost = this.DeliveryCost;

    return snapshot;
}

public static void CreateBasketFrom(BasketSnapshot snapshot)
{
    return new Basket(snapshot);
}
}
```

Реализация соответствующего репозитория могла бы выглядеть, как показано в листинге 21.12.

Листинг 21.12. Реализация репозитория корзины

```
namespace Infrastructure.Persistence
{
    public class BasketRepository : IBasketRepository
    {
        // ....

        public Basket FindBy(Guid id)
        {
            Basket basket;

            BasketDataModel
            basketDataModel = FindDataModelBy(id);
```

```

    if (basketDataModel != null)
    {
        var basketSnapshot = ConvertToBasketSnapshot(basketDataModel);

        basket = Basket.CreateBasketFrom(basketSnapshot);
    }

    return basket;
}
}
}

```

Снимок состояния можно также напрямую отобразить в модель данных, если использовать фреймворк, позволяющий сделать это, но не позволяющий отобразить полную предметную модель.

Потоки событий

Другой подход к сохранению предметной модели — использовать поток событий. Потоки событий напоминают шаблон «Хранитель», только вместо моментального снимка состояния агрегата репозиторий сохраняет все события, возникшие в агрегате, в форме предметных событий. Обработчик принимает события из предметной области и отображает их в предметную модель. И снова необходим фабричный метод, который восстановит агрегат по зафиксированным событиям. На рис. 21.5 изображено графическое представление этого подхода. Подробнее о потоках событий рассказывается в главе 22 «Регистрация событий».

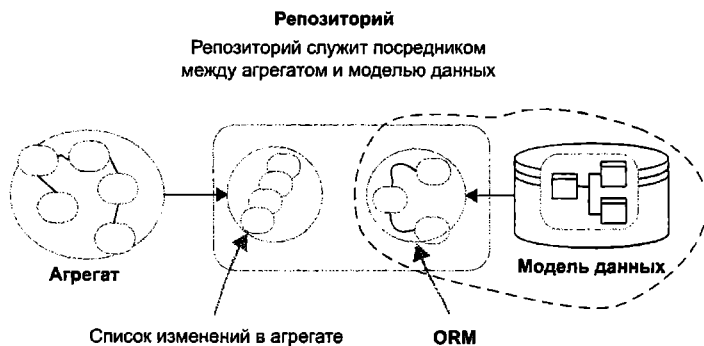


Рис. 21.5. Сохранение событий, возникших в агрегате

Прагматизм

Во всех стратегиях сохранения предметной модели следует проявлять здоровый прагматизм. Чистая предметная модель не должна зависеть от стратегии сохранения, в том смысле что на ней не должны сказываться любые изменения, необходимые для использования фреймворка сохранения. Однако чистота хороша в тео-

рии, на практике же она труднодостижима, и иногда приходится принимать более прагматичное решение.

Если корпоративные требования ограничивают выбор типов хранилищ или фреймворков, вам, возможно, придется пойти на компромиссы, когда дело дойдет до физического сохранения предметных объектов. Выбирайте стратегии, лучшие всего подходящие для доступных вам фреймворков и хранилищ данных. Старайтесь не бороться со своим фреймворком, а выяснить, что нужно изменить в предметной модели, с учетом достижения наибольшей производительности и масштабируемости. Но не следует задумываться об этом с самого начала; начинайте создавать чистую предметную модель и только потом ищите практическое решение, если это понадобится. Не позволяйте техническим особенностям фреймворков управлять проектированием; размышляйте в терминах агрегатов и ясно оформленных процедур извлечения данных, не думая об особенностях выполнения запросов.

Репозиторий — четко определенный контракт

Контракт репозитория — это не только CRUD-интерфейс. Он дополняет предметную модель и определяется в терминах, понятных специалистам в предметной области. Репозиторий должен строиться исходя из потребностей сценариев использования приложения, а не с позиции CRUD-подобной организации доступа к данным. Клиентом контракта является прикладной уровень, он извлекает агрегаты из репозитория и делегирует им выполнение основной работы. Но определяются контракты на уровне предметной модели, которая, впрочем, почти не использует их, за исключением редких случаев, когда это требуется в предметных службах.

Репозиторий — это не объект, а процедурная граница и четко определенный контракт, который требует тщательно подбирать имена методов, как это делают объекты в предметной модели. Контракт репозитория должен быть специализированным и однозначно раскрывать намерения, понятные специалистам в предметной области.

Рассмотрим интерфейс репозитория в листинге 21.13. Он дает клиенту возможность запрашивать некоторые предметные объекты. Это гибкий и универсальный контракт, но он ничего не сообщает о потребностях или целях, стоящих за попыткой получить коллекции агрегатов. Чтобы понять, как использовать эти методы, разработчик должен просмотреть весь код и разобраться, с какой целью выполняются запросы, не говоря уже о возможности их оптимизации. Поскольку контракт неспециализирован и допускает слишком широкую интерпретацию, он практически бесполезен и как граница, и как интерфейс.

Листинг 21.13. Интерфейс репозитория с методами запроса, открытыми для расширения

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
```

```
Customer FindBy(Guid id);  
IEnumerable<Customer> FindAllThatMatch(Query query);  
IEnumerable<Customer> FindAllThatMatch(String hql);  
void Add(Customer customer);  
}  
}
```

Теперь взгляните на другое определение контракта репозитория в листинге 21.14. Этот контракт ясно и четко сообщает, как будут использоваться агрегаты. Чрезмерно универсальные методы были заменены двумя специализированными методами с именами, отражающими намерения на языке, понятном специалистам в предметной области. Здесь применен принцип «говори, а не спрашивай», вынуждающий репозиторий выполнить всю основную работу, скрытую за абстракцией метода запроса. Теперь совершенно ясно, как можно оптимизировать эти запросы в реализации инфраструктуры.

Листинг 21.14. Интерфейс репозитория с говорящими именами методов

```
namespace DomainModel  
{  
    public interface ICustomerRepository  
    {  
        Customer FindBy(Guid id);  
        IEnumerable<Customer> FindAllThatAreDeactivated();  
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();  
        void Add(Customer customer);  
    }  
}
```

Репозиторий — это контракт между предметной моделью и хранилищем. Он должен определяться только в предметных терминах и без какого-либо намека на конкретный фреймворк. Определите цель и сделайте ее явной; не рассматривайте контракт репозитория как объектно-ориентированный код.

Управление транзакциями и единицы работы

Управление транзакциями главным образом осуществляется в уровне прикладных служб. Но так как управление транзакциями и репозитории тесно связаны друг с другом, имеет смысл коснуться этой темы здесь (более подробно об управлении транзакциями рассказывается в следующей главе). Любой репозиторий управляет лишь единственной коллекцией корней агрегатов, тогда как для выполнения сценариев использования может потребоваться изменить множество агрегатов разных типов.

Транзакция представляет собой единицу работы. Роль шаблона «Единица работы» (unit-of-work) состоит в том, чтобы гарантировать согласованное изменение всех агрегатов, вовлеченных в решение бизнес-задачи. Как только все необходимые изменения будут внесены, реализация шаблона «Единица работы» осуществ-

влет запись в хранилище изменений, выполненных в рамках транзакции. Чтобы обеспечить целостность данных в случае появления каких-либо проблем при передаче изменений в хранилище, все изменения откатываются до исходного состояния — и тем самым гарантируется их сохранность в допустимом состоянии.

В NHibernate шаблон «Единица работы» реализует объект сеанса. В листинге 21.15 показана прикладная служба, которая использует два репозитория для вычисления оценки лояльности. Управление транзакцией осуществляется с помощью объекта сеанса из фреймворка NHibernate. Никакие изменения не сохраняются в базе данных, пока запущенная транзакция единицы работы не будет подтверждена.

Листинг 21.15. Реализация шаблона «Единица работы» с использованием NHibernate

```
namespace ApplicationLayer
{
    public class LoyalPointsAccumulator
    {
        private ILoyaltyAccountRepository _loyaltyAccountsRepo;
        private IOrderRepository _ordersRepo;
        private LoyaltyPointsCalculator _pointsCalculator;
        private ISession _session;

        public LoyalPointsAccumulator(
            ILoyaltyAccountRepository loyaltyAccountsRepo,
            IOrderRepository ordersRepo,
            LoyaltyPointsCalculator pointsCalculator,
            ISession session)
        {
            _loyaltyAccountsRepo = loyaltyAccountsRepo;
            _ordersRepo = ordersRepo;
            _pointsCalculator = pointsCalculator;
            _session = session;
        }

        public void CalculatePointsFor(Guid orderId)
        {
            using(var transaction = _session.BeginTransaction())
            {
                var order = _ordersRepo.FindMatching(orderId)
                var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

                var points = _pointsCalculator.calculateFor(order);

                account.Add(points, orderId);

                order.Earns(points);

                transaction.Commit();
            }
        }
    }
}
```

На уровне прикладной службы не требуется абстрагироваться от используемого фреймворка сохранения, потому что это не дает никаких преимуществ. Здесь явно сообщается, что приложение использует фреймворк NHibernate. В данном случае только предметная модель должна оставаться независимой от выбора механизма хранения. Прикладной службе не требуется абстрагироваться от фреймворка, потому что она руководит выполнением инфраструктурной и предметной логики.

Чтобы внести чуть больше ясности, на рис. 21.6 показано, как репозитории взаимодействуют с объектом сеанса (реализующим шаблон «Единица работы» в NHibernate).

Чтобы гарантировать применение всех изменений, выполненных в рамках одной транзакции, репозитории должны использовать тот же экземпляр сеанса, который используется прикладной службой. Это достигается за счет использования фабрики или контейнера инверсии управления (*inversion of control container*) для создания прикладной службы, о чем подробно рассказывается в следующей главе. Экземпляр репозитория получает объект сеанса через параметр конструктора, как показано в листинге 21.16.

Листинг 21.16. Внедрение объекта сеанса NHibernate в параметр конструктора репозитория

```
namespace Infrastructure.Persistence
{
    public class OrderRepository : IOrderRepository
    {
        private ISession _session;

        public OrderRepository(ISession session)
        {
            _session = session;
        }

        public void Add(Order order)
        {
            _session.Save(order);
        }

        public Order FindBy(Guid id)
        {
            return _session.Load<Order>(id);
        }
    }
}
```

Также для передачи экземпляра сеанса можно использовать прием внедрения в параметр метода записи (*setter-based injection*). Это еще больше прояснит происходящее. В листинге 21.17 показано, как можно изменить прикладную службу, чтобы использовать прием передачи сеанса через параметр метода записи.

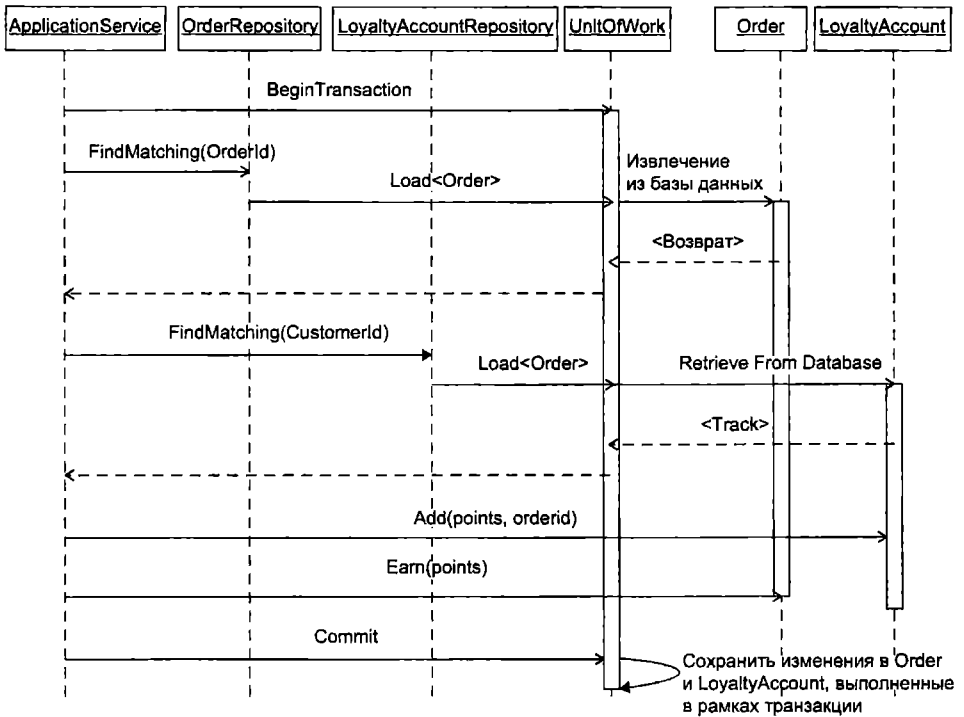


Рис. 21.6. Шаблон «Единица работы»

Листинг 21.17. Внедрение объекта сеанса NHibernate через общедоступное свойство репозитория

```

namespace ApplicationLayer
{
    public class LoyaltyPointsAccumulator
    {
        // ....

        public void CalculatePointsFor(Guid orderId)
        {
            using(var transaction = _session.BeginTransaction())
            {
                _ordersRepo.EnlistIn(_session);
                _loyaltyAccountsRepo.EnlistIn(_session);

                var order = _ordersRepo.FindMatching(orderId)
                var account = loyaltyAccountsRepo.FindMatching(order.CustomerId)

                var points = _pointsCalculator.calculateFor(order);
                account.Add(points, orderId);
                order.Earns(points);
            }
        }
    }
}
  
```

```

        transaction.Commit();
    }
}
}
}

```

Так как шаблон «Единица работы» никак не затрагивает предметную область и решает задачи исключительно технического характера, нет никакого смысла включать метод `EnlistIn` непосредственно в интерфейс репозитория. Вместо этого можно определить отдельный интерфейс, включить в него метод `EnlistIn` и реализовать в репозитории проверку существования объекта сеанса перед выполнением любых операций (см. листинг 21.18).

Листинг 21.18. Использование дополнительного интерфейса для разделения зон ответственности единицы работы и репозитория

```

namespace Infrastructure.Persistence
{
    public interface IEnlistInAUnitOfWork
    {
        void EnlistIn(ISession session);
    }
}

namespace Infrastructure.Persistence
{
    public class OrderRepository : IOrderRepository, IEnlistInAUnitOfWork
    {
        private ISession _session;

        public void EnlistIn(ISession session)
        {
            _session = session;
        }

        public void Add(Order order)
        {
            _session.Save(order);
        }

        public Order FindBy(Guid id)
        {
            return _session.Load<Order>(id);
        }
    }
}

```

Сохранять или не сохранять

Репозиторий должен представлять собой коллекцию агрегатов и действовать подобно коллекциям .NET, полностью скрывая особенности работы механизма хра-

нения. Коллекции .NET хранят ссылки на объекты, поэтому никакие изменения в любом из объектов не требуют явного обновления самой коллекции. Из этого следует, что коллекция не обязана иметь метод сохранения или обновления; в идеале репозиторий должен действовать аналогично. Однако имитация интерфейса коллекций в значительной степени зависит от возможностей фреймворка сохранения. Фреймворк, не поддерживающий возможности автоматического определения изменений, должен экспортировать метод сохранения, а прикладная служба, координирующая операции в предметной области, должна явно сохранять агрегаты после их изменения. Если вы решите обойтись без использования фреймворка сохранения, вам придется изрядно потрудиться. Лучше будет рассмотреть возможные варианты, которые избавят вас от рутины, связанной с сохранением вручную.

Фреймворки сохранения, автоматически определяющие изменения в предметных объектах

Фреймворки ORM, такие как NHibernate, способны автоматически определять изменения в объектах, извлекаемых с их помощью. Так как изменения определяются автоматически, клиент репозитория (обычно прикладная служба) не должен явно сохранять эти объекты. Это означает, что нет нужды включать метод `Save` в контракт репозитория, потому что изменения передаются в хранилище автоматически, по завершении единицы работы. Взгляните на листинг 21.19, где представлены контракт репозитория и пример прикладной службы. Здесь используется абстракция шаблона «Единица работы», которая может быть реализована посредством фреймворка сохранения или вручную.

Листинг 21.19. Реализация репозитория на основе NHibernate, не требующая метода `Save`

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        IEnumerable<Customer> FindAllThatAreDeactivated();
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
        void Add(Customer customer);
    }
}

namespace ApplicationLayer
{
    public class LoyalPointsAccumulator
    {
        // ....

        public void CalculatePointsFor(Guid orderId)
        {
```

```

        var order = _ordersRepo.FindMatching(orderId)
        var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

        var points = _pointsCalculator.calculateFor(order);

        account.Add(points, orderid);

        order.Earns(points);

        uow.Commit();
    }
}

```

Для выявления изменений в предметных объектах фреймворк сохранения либо создает копии агрегатов, извлекаемых в процессе чтения, и сравнивает фактические агрегаты с их копиями по завершении единицы работы, чтобы определить, что сохранять, либо возвращает прокси-объект агрегата и отслеживает факт изменений с его помощью. После вызова метода подтверждения единицы работы фреймворк выясняет, производились ли изменения, и затем генерирует SQL-запрос, необходимый для сохранения изменений в модели данных.

Необходимость явно сохранять агрегаты

Если для отображения в модель данных используется фреймворк, не поддерживающий автоматическое определение изменений (какой-либо микро-фреймворк ORM), или используется низкоуровневый механизм ADO.NET, вам придется добавить в контракт репозитория метод сохранения и вызывать его в прикладной службе в случае изменения предметных объектов. В листинге 21.20 представлен интерфейс репозитория с методом `Save` и показано, как он используется прикладной службой. Вы все еще можете вызывать единицу работы, чтобы гарантировать применение всех изменений в рамках одной транзакции.

Листинг 21.20. Реализация репозитория с методом `Save`

```

namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        IEnumerable<Customer> FindAllThatAreDeactivated();
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
        void Add(Customer customer);
        void Save(Customer customer);
    }
}

namespace ApplicationLayer
{
    public class LoyaltyPointsAccumulator
    {

```

```
// ....

public void CalculatePointsFor(Guid orderId)
{
    var order = _ordersRepo.FindMatching(orderId)
    var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

    var points = _pointsCalculator.calculateFor(order);

    account.Add(points, orderId);

    order.Earns(points);

    _ordersRepo.Save(order);
    _loyaltyAccountsRepo.Save(account);

    uow.Commit();
}
}
```

НЕ ИСПОЛЬЗУЙТЕ ФЛАГИ ИЗМЕНЕНИЯ В ПРЕДМЕТНЫХ ОБЪЕКТАХ

У кого-то может возникнуть соблазн добавить в предметный объект флаг, сообщающий, что объект изменился. Проблема такого решения состоит в загрязнении предметной модели аспектами, связанными с задачами сохранения. Оставьте определение изменений на долю единицы работы и репозитория. Если вы не используете инструменты ORM, не волнуйтесь — в конце главы будет представлен пример реализации агрегатов и репозитория, не зависящих от особенностей механизма хранения.

Репозиторий как предохранительный слой

При работе в унаследованном окружении с существующими хранилищами репозиторий может помочь сохранить чистоту предметной модели, выполняя роль предохранительного слоя и позволяя формировать модель без оглядки на любые сложности базовой инфраструктуры.

Как уже рассказывалось ранее, модель данных и предметная модель могут иметь существенные различия. Модель данных может быть разбросана по множеству таблиц или даже баз данных. Кроме того, может использоваться множество разнотипных хранилищ, таких как обычные файлы, веб-службы и реляционные или документоориентированные хранилища. Какое бы хранилище ни использовалось, оно не должно влиять на форму предметной модели. Репозитории отображают данные в агрегаты, а не в таблицы; например, в одной таблице может храниться множество сущностей, как рассказывалось в главе 16, где было показано, как ис-

пользовать явные реализации для разных состояний. Репозитории могут также сохранять модель более чем в одном хранилище, как показано на рис. 21.7.

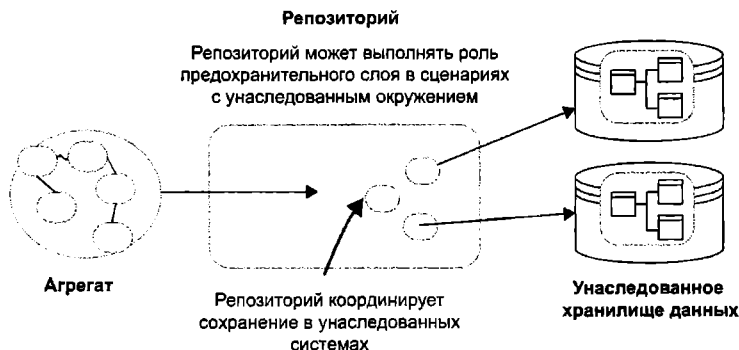


Рис. 21.7. Репозиторий выполняет роль предохранительного слоя

Другие области ответственности репозитория

Помимо извлечения и сохранения агрегатов, репозиторий может экспортировать дополнительную информацию о своей коллекции предметных объектов. Например, репозиторий может экспортировать число агрегатов в коллекции, как показано в листинге 21.21.

Листинг 21.21. Репозиторий клиентов с методами коллекции

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        IEnumerable<Customer> FindAllThatAreDeactivated();
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
        void Add(Customer customer);
        int Count();
    }
}
```

Генерирование идентификаторов сущностей

Если база данных или другая инфраструктурная служба управляет раздачей идентификаторов, эту операцию можно абстрагировать за фасадом репозитория и экспортировать для использования в прикладных службах. Создание уникальных идентификаторов можно экспортировать через метод в интерфейсе, как показано в листинге 21.22.

Листинг 21.22. Абстракция создания идентификаторов

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
        IEnumerable<Customer> FindAllThatAreDeactivated();
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
        void Add(Customer customer);
        Guid GenerateId();
    }
}

namespace ApplicationLayer
{
    public class CustomerRegistration
    {
        private ILoyaltyAccountRepository _loyaltyAccountRepo;
        private ICustomerRepository _customerRepo;
        private ISession _session

        public CustomerRegistration(
            ILoyaltyAccountRepository loyaltyAccountRepo,
            ICustomerRepository customerRepo,
            ISession session)
        {
            _loyaltyAccountRepo = loyaltyAccountRepo;
            _customerRepo = customerRepo;
            _session = session;
        }

        public void Register(CustomerDetail details)
        {
            using(var transaction = _session.BeginTransaction())
            {
                var newCustomerId = _customerRepo.GenerateId();
                var customer = CustomerRegistration.CreateFrom(
                    details, newCustomerId);

                _customerRepo.Add(customer);

                var loyaltyAccount = new LoyaltyAccount(newCustomerId);
                _loyaltyAccountRepo.Add(loyaltyAccount);

                transaction.Commit();
            }
        }
    }
}
```

Идентификаторы могут также назначаться репозиторием в процессе сохранения. При этом за основу обычно принимаются идентификаторы, генерируемые хранилищем данных. В листинге 21.23 приводится фрагмент XML-документа, описывающего отображение данных для NHibernate. Атрибут `class` вложенного тега `generator` определяет, как должен генерироваться идентификатор сущности. В данном случае значение `"native"` сообщает, что должны использоваться идентификаторы, автоматически генерируемые базой данных. Далее вы увидите другие варианты генерирования идентификаторов для сущностей. Атрибут `unsaved-value` используется фреймворком NHibernate для сравнения с идентификатором сущности, чтобы определить, является объект хранимым или временным (не требующим сохранения).

Листинг 21.23. Фрагмент XML-документа с описанием отображения для NHibernate

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
    namespace="DomainModel" assembly="DomainModel">

    <class name="Order" table="Orders" lazy="false" >

        <id name="Id" column="OrderId" type="int" unsaved-value="0">
            <generator class="native" />
        </id>

        // ....

    </class>
</hibernate-mapping>
```

Не беспокойтесь, если вы не знакомы с NHibernate, потому что ближе к концу главы будет продемонстрировано еще несколько примеров реализации данных шаблонов на практике.

Обобщенные сведения о коллекциях

Помимо числа агрегатов в коллекции, иногда бывает желательно иметь некоторые другие, обобщенные сведения о содержимом коллекции, чтобы не приходилось извлекать каждый агрегат и выполнять подсчет вручную. В листинге 21.24 определяется объект-значение `Summary`, позволяющий получить количество сущностей-клиентов разного вида. Для многочисленных запросов и вычислений, которые лучше работают с низкоуровневыми данными, представление обобщенных сведений может оказаться чрезвычайно мощным инструментом.

Листинг 21.24. Репозиторий с методами для получения обобщенной информации

```
namespace DomainModel
{
    public interface ICustomerRepository
    {
        Customer FindBy(Guid id);
    }
}
```



```

        IEnumerable<Customer> FindAllThatAreDeactivated();
        IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
        void Add(Customer customer);
        Summary Summary();
    }
}

namespace DomainModel
{
    public class Summary
    {
        public int CustomersCount { get; set; }
        public int DeactivatedCustomers { get; set; }
        public int CustomersOverAllowedCredit { get; set; }
    }
}

```

Одновременный доступ

Когда множество пользователей одновременно изменяют состояние предметного объекта, важно, чтобы они работали с последней версией агрегата и их изменения не затирали другие изменения, о которых они не знают. Существует две формы управления одновременным доступом: оптимистическая и пессимистическая. Оптимистическая форма управления предполагает отсутствие проблем при одновременном изменении состояния предметных объектов несколькими пользователями и известна как «побеждает последний». Для некоторых систем это вполне разумное поведение; однако когда требуется, чтобы объекты находились в согласованном состоянии при извлечении из базы данных, необходимо использовать пессимистическую форму управления.

Пессимистическая форма управления одновременным доступом имеет множество разновидностей — от блокировки таблицы данных на время извлечения записи до сохранения копии содержимого предметного объекта и сравнения ее с версией в хранилище перед сохранением. Многие фреймворки сохранения используют номер версии, чтобы проверить, изменялся ли предметный объект с момента последнего извлечения из базы данных. После обновления, но перед подтверждением транзакции номер версии сущности сравнивается с номером версии в базе данных. Такая проверка помогает убедиться, что сущность не изменялась с момента ее извлечения.

Фреймворк NHibernate, например, поддерживает несколько вариантов оптимистического управления одновременным доступом. Один из них основан на использовании тега `version` в XML-документе с описанием отображения, как показано в листинге 21.25.

Листинг 21.25. Сущность с полем для хранения номера версии

```

namespace DomainModel
{
    public class Product
    {
        private int _version;
    }
}

```

```

        // ...
    }
}

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="DomainModel" assembly="DomainModel">

    <class name="Product" table="Products" lazy="false" >

        <id name="Id" column="Id" type="int" unsaved-value="0">
            <generator class="native" />
        </id>

        // ....

        <version name="Version" column="Version"/>
    </class>
</hibernate-mapping>

```

Свойство `Version` устанавливается в момент извлечения сущности `Product` из хранилища. Если свойство `Version` кажется вам неуместным в сущности `Product` в связи с тем, что в моделируемой предметной области номер версии не является атрибутом продукта, можно использовать суперкласс `Entity`, как показано в листинге 21.26 (см. также главу 16) или возвращать прокси-версию сущности `Product` и включать номер версии в нее.

Листинг 21.26. Свойство `Version` в базовом классе `Entity`

```

namespace Infrastructure
{
    public abstract class Entity
    {
        public int VERSION {get; private set; };
    }
}

namespace DomainModel
{
    public class Product : Entity
    {
        // ...
    }
}

```

В листинге 21.27, где представлен фрагмент определения класса прикладной службы, можно видеть, что когда сущность `Product` сохраняется в базе данных, версия измененной сущности включается в предложение `where`. `NHibernate` сравнит два значения и возбудит исключение `StaleObjectStateException`, если объект `Product` изменился и ваше изменение оказалось вторым. В этом случае программа должна

известить пользователя, что сущность `Product` изменилась или была удалена другим пользователем после ее извлечения и ее обновление оказалось невозможным.

Листинг 21.27. Прикладная служба обрабатывает исключения, возникающие в результате одновременного доступа

```
namespace ApplicationLayer
{
    public class Pricing
    {
        private IProductRepository _productRepository;
        private ISession _session;

        public Pricing(IProductRepository productRepo, ISession session)
        {
            _productRepository = productRepo;
            _session = session;
        }

        public void SetPrice(
            UpdatedProductPriceInformation updatedProductInformation)
        {
            try
            {
                using(var transaction = _session.BeginTransaction())
                {
                    var product = _productRepo
                        .FindBy(updatedProductInformation.ProductId);

                    product.SetPriceTo(updatedProductInformation.NewPrice);

                    transaction.Commit();
                }
            }
            catch (StaleObjectStateException ex)
            {
                // Продукт изменился между моментами его извлечения
                // и подтверждения транзакции
            }
        }
    }
}
```

Программный код в листинге 21.27 получит исключение, если в короткий промежуток времени между извлечением из хранилища и сохранением изменений продукт был изменен кем-то другим. Такое вполне возможно, пока пользователь рассматривает продукт на своем компьютере в браузере. В данной ситуации продукт мог быть изменен, пока пользователь просматривает данные, и он мог бы сохранить свои изменения, затерев изменения, сделанные другим пользователем. Чтобы предотвратить это, вместе с моделью представления в пользовательский интерфейс должен передаваться номер версии и тот же номер должен возвращаться в `UpdatedProductPriceInformation`. Исправленный код в листинге 21.28 про-

веряет совпадение номера версии измененного пользователем продукта с номером версии, полученным из репозитория. Если они не совпадают, возбуждается исключение. В этом случае пользователь должен запросить обновленную версию данных и просмотреть их еще раз, прежде чем изменить цену.

Листинг 21.28. Проверка свойства Version базового класса Entity в пользовательском интерфейсе

```
namespace ApplicationLayer
{
    public class Pricing
    {
        private IProductRepository _productRepo;
        private ISession _session;

        public Pricing(IProductRepository productRepo, ISession session)
        {
            _productRepo = productRepo;
            _session = session;
        }

        public void SetPrice(
            UpdatedProductPriceInformation updatedProductInformation)
        {
            try
            {
                using(var transaction = _session.BeginTransaction())
                {
                    var product = _productRepo
                        .FindBy(updatedProductInformation.ProductId);

                    if (updatedProductInformation.Version != product.Version)
                        throw new ProductHasBeenUpdatedSinceViewingException();

                    product.SetPriceTo(updatedProductInformation.NewPrice);

                    transaction.Commit();
                }
            }
            catch (StaleObjectStateException ex)
            {
                Throw New StaleDataUpdateReportBeforeSendingCommand();
            }
        }
    }
}
```

Контрольные проверки

Если модель данных требует включения метаданных, не имеющих смысла в предметной области, обеспечить соответствие этим требованиям можно с помощью репозитория. Часто изменяемые объекты должны включать дату последнего изме-

нения, не несущую смысловой нагрузки для предметной области. Эти метаданные можно добавлять в реализации репозитория.

Контрольные проверки и регистрация (журналирование) также могут осуществляться посредством репозитория. Если вы удаляете агрегат, репозиторий мог бы зарегистрировать это событие в журнале и, возможно, сохранить там же содержимое агрегата или сводную информацию о нем.

Антишаблоны реализации репозитория

Как и для любого шаблона, существует множество nereкомендуемых приемов, которых следует избегать. Такие приемы часто называют антишаблонами.

Антишаблон: поддержка универсальных запросов

Репозиторий — это четкий контракт между предметной моделью и механизмом хранения. Экспортируя универсальные методы «все в одном» для выполнения специализированных запросов, вы ослабляете этот уровень абстракции. Взгляните на контракт в листинге 21.29.

Листинг 21.29. Экспортирование универсальных методов `FindBy` вместо более четких и специализированных методов

```
public interface ICustomerRepository
{
    void Add(Customer customer);
    Customer FindBy(Guid Id);
    IEnumerable<Customer> FindBy(CustomerQuery query);
}
```

Метод `FindBy` принимает некоторый запрос, возвращающий коллекцию клиентов, возможно, в форме некоторой спецификации. Проблема здесь в том, что пользователь может передать методу любой запрос, а это означает необходимость оптимизации всех вариантов выборки. Подобные методы ослабляют контракт репозитория и полностью скрывают цель обращения к репозиторию. Репозиторий не должен быть открытым для расширения; он имеет ограниченную и процедурную природу. Вместо этого следует выбирать такие имена для методов, чтобы можно было писать специализированный код, пригодный для оптимизации и настройки под фреймворк сохранения. Новые добавляемые методы могут настраиваться, и для их реализации может быть выбрана наиболее подходящая стратегия извлечения.

Экспортирование интерфейса `IQueryable` в репозитории — еще одна разновидность поддержки специализированных методов запроса. Интерфейс `IQueryable` — чрезвычайно гибкий шаблон доступа к данным, но он может спровоцировать проникновение модели данных в предметную модель. Однако вполне допустимо использовать интерфейс `IQueryable` за границами репозитория или непосредственно для доступа к модели данных с целью составления отчетов, как показано в листинге 21.30.

Листинг 21.30. Интерфейс репозитория клиентов

```
public interface ICustomerRepository
{
    void Add(Customer customer);
    Customer FindBy(Guid Id);
    IQueryable<Customer> Customers();
}
```

Антишаблон: отложенная загрузка

Агрегаты должны конструироваться на основе инвариантов и включать все свойства, необходимые для соответствия этим инвариантам. Поэтому агрегаты должны загружаться целиком или не загружаться вовсе. Если вы пользуетесь реляционной базой данных и инструментом ORM, у вас есть возможность организовать отложенную загрузку некоторых свойств предметных объектов, то есть отложить загрузку компонентов агрегата, которые не нужны в данный момент. Проблема, однако, в том, что если есть возможность загрузить только часть агрегата, значит, вы, вероятно, неправильно определили его границы. Кроме того, если шаблоны извлечения данных запрашивают ваши агрегаты, значит, вы, возможно, сконструировали их, опираясь на потребности пользовательского интерфейса, а не основываясь на бизнес-правилах.

Антишаблон: использование репозитория для составления отчетов

Потребности пользовательского интерфейса и бизнес-логики отличаются существенно. Очень часто наблюдается несоответствие между информацией, необходимой для отображения на экране, и данными, содержащимися в единственном корне агрегата. Если для отображения потребуется только одно или два свойства, придется извлекать агрегат целиком. Хуже того, вместе с предметными объектами может потребоваться отображать дополнительную информацию, не несущую смысла для предметных объектов, к которым она присоединяется.

Кроме того, при создании для отображения на экране такой модели представления, которая объединяет несколько корней агрегатов, может понадобиться отбросить их все и выбрать только нужную информацию.

Выполнение отчетов с использованием модели транзакций может ухудшить производительность и поставить под угрозу целостность модели из-за дополнительных свойств, необходимых для отображения в пользовательском интерфейсе. Намного лучше использовать возможности фреймворка для выполнения прямых запросов к хранилищу; это может быть то же хранилище, что используется транзакциями, или другое, денормализованное хранилище.

Не создавайте абстракции фреймворка сохранения, запрашивая модель данных для нужд составления отчетов. Подробнее об этом рассказывается в следующей главе.

Реализации репозиториев

В этом разделе вы познакомитесь с некоторыми реализациями репозиториев, основанными на популярных фреймворках сохранения для .NET, а именно:

- NHibernate
- RavenDB
- Entity Framework
- Dapper Micro ORM

Для примеров будет использоваться небольшая предметная модель, основанная на логике интернет-аукциона eBay. Эта модель изображена на рис. 21.8. Мы не будем следовать методологии разработки «сначала тест», потому что цель этих примеров — демонстрация возможных реализаций, а не обсуждение проекта предметной модели. Учитывая это, мы будем конструировать решения так, чтобы вы могли изучить их и воссоздать у себя.

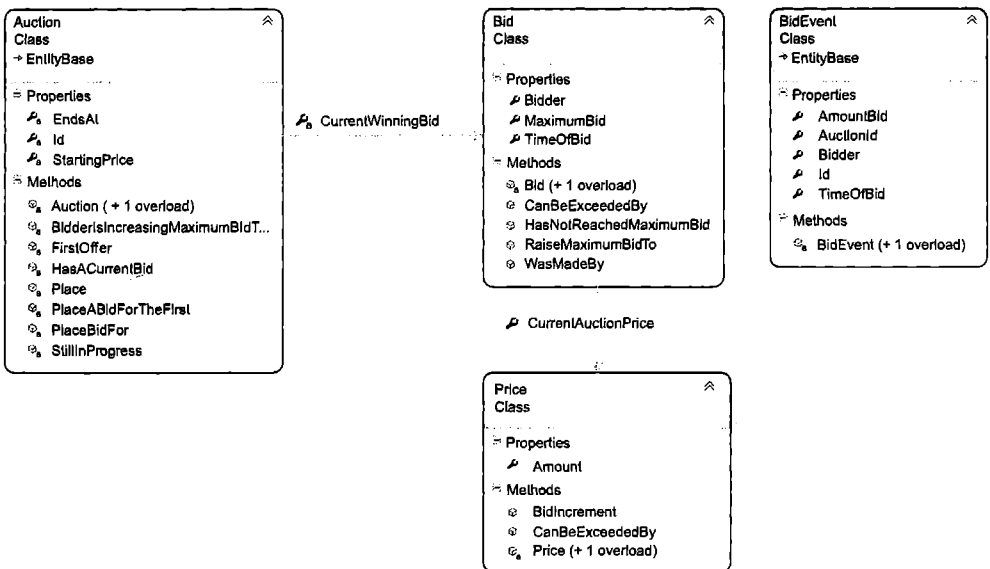


Рис. 21.8. Предметная модель

Модель представляет логику работы механизма ставок на eBay:

- Когда участник торгов делает ставку, он вводит максимальную сумму, которую готов заплатить за лот. Однако фактически устанавливается цена, минимально необходимая, чтобы перебить предыдущую ставку или начальную цену.
- Когда второй участник делает ставку, ставка предыдущего участника автоматически увеличивается до максимального значения либо до уровня, достаточного, чтобы перебить ставку текущего участника.

- Если второй участник предложил ставку, превышающую максимальную сумму предыдущего участника, предыдущий участник уведомляется, что его ставка была превышена.
- Если второй участник предложил ту же максимальную сумму, что и предыдущий, предыдущий участник остается победителем, потому что он первый предложил эту сумму.

Все фреймворки сохранения, которые будут использоваться, оказывают не большое влияние на предметную модель. Примеры ниже демонстрируют, что вполне возможно создать предметную модель, независимую от выбранного фреймворка.

Фреймворк сохранения способен отобразить предметную модель в модель данных без компромиссов

В первых двух примерах реализации репозитория используются NHibernate и RavenDB. Оба фреймворка позволяют отобразить предметную модель непосредственно в модель данных без компромиссов или с незначительными компромиссами.

Пример для NHibernate

NHibernate — это .NET-версия популярного открытого фреймворка Hibernate для Java. Он развивается уже несколько лет и давно доказал свою надежность. NHibernate — это фреймворк объектно-реляционного отображения (ORM). Одна из замечательных особенностей NHibernate — поддержка независимости от механизма хранения; это означает, что предметные объекты не должны наследовать какие-либо базовые классы или реализовывать интерфейсы фреймворка. NHibernate использует экземпляр `ISession`, который является реализацией шаблона «Единица работы». `ISession` также используется для выполнения запросов к базе данных. Он действует как диспетчер хранилища и шлюз в базу данных, позволяя извлекать, сохранять, удалять и добавлять сущности. NHibernate реализует множество способов отображения предметных объектов в таблицы базы данных. Наиболее популярным является способ на основе конфигурационного XML-файла, но также имеется возможность реализовать отображение с помощью атрибутов и сменного кода. В этом примере используется подход с использованием XML-отображения.

Предметная модель в этом примере не имеет общедоступных свойств и является полностью инкапсулированной. NHibernate способен сохранять такие закрытые модели.

Настройка решения

Создайте в Visual Studio новое пустое решение с именем `DDDDPP.Chap21.NHibernateExample`, добавьте в него библиотеку классов с именем `DDDDPP.Chap21`.

`NHibernateExample.Application` и консольное приложение с именем `DDDDPP.Chap21.NHibernateExample.Presentation`. В проекте `Presentation` добавьте ссылку на библиотеку `Application`. Далее, создайте следующие папки в проекте `DDDDPP.Chap21.NHibernateExample.Application`:

- `Application`
- `Infrastructure`
- `Model`

Можете удалить файл `Class1.cs`, созданный автоматически, потому что он нам не понадобится. Получившееся решение должно выглядеть, как показано на рис. 21.9.

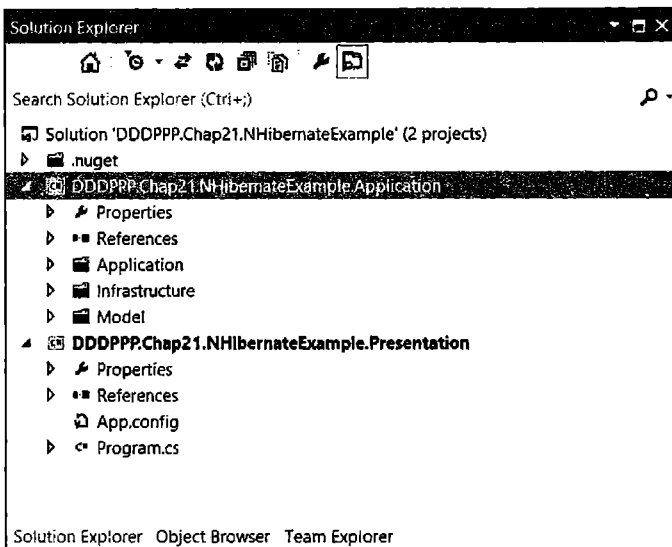


Рис. 21.9. Структура проекта в Visual Studio

Прежде чем собрать приложение, нужно добавить ссылки на сборки `NHibernate`. Откройте окно диспетчера пакетов `NuGet` и установите `NHibernate`, как показано на рис. 21.10.

Создание модели

Первым делом нужно создать модель. Реализация будет использовать предметные события, поэтому нужно определить инфраструктуру поддержки событий. Мы воспользуемся инфраструктурой, сконструированной в главе 18 «События предметной области». Добавьте в папку `Infrastructure`, в библиотеке классов `Application`, новый класс `DomainEvents`, представленный в листинге 21.31. Подробное описание этого класса можно найти в главе 18.

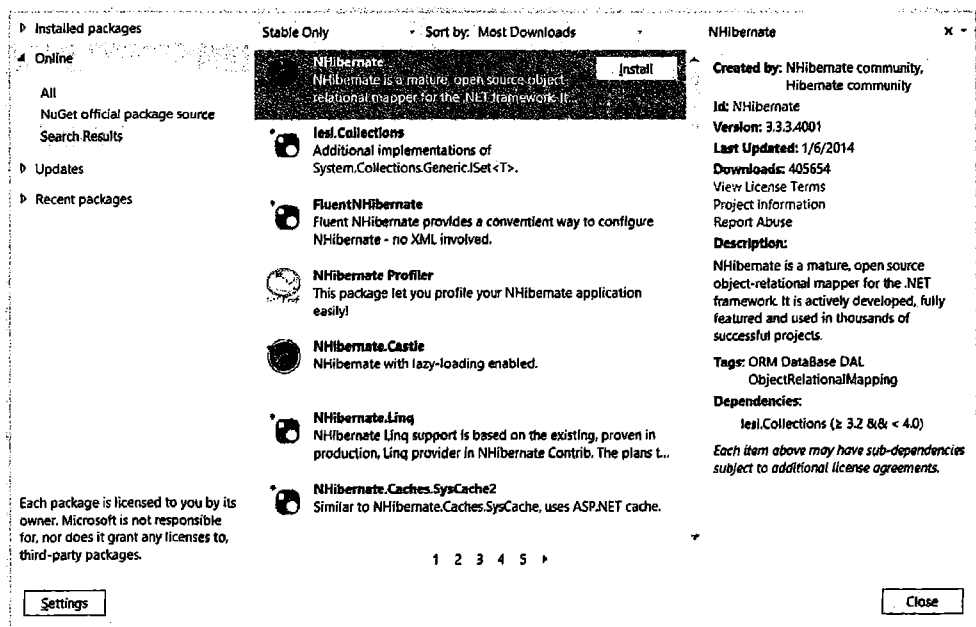


Рис. 21.10. Диспетчер пакетов NuGet

Листинг 21.31. Класс инфраструктуры поддержки предметных событий

```

using System;
using System.Collections.Generic;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    /// <summary>
    /// Класс поддержки предметных событий из
    /// http://www.udidahan.com/2008/08/25/domain-events-take-2/
    /// </summary>
    public static class DomainEvents
    {
        [ThreadStatic]
        private static List<Delegate> _actions;
        private static List<Delegate> Actions
        {
            get
            {
                if (_actions == null)
                {
                    _actions = new List<Delegate>();
                }
                return _actions;
            }
        }
    }
}

```

```

public static IDisposable Register<T>(Action<T> callback)
{
    Actions.Add(callback);

    return new DomainEventRegistrationRemover(() =>
        Actions.Remove(callback));
}

public static void Raise<T>(T eventArgs)
{
    foreach (Delegate action in Actions)
    {
        Action<T> typedAction = action as Action<T>;
        if (typedAction != null)
        {
            typedAction(eventArgs);
        }
    }
}

private sealed class DomainEventRegistrationRemover : IDisposable
{
    private readonly Action _callOnDispose;

    public DomainEventRegistrationRemover(Action toCall)
    {
        _callOnDispose = toCall;
    }

    public void Dispose()
    {
        _callOnDispose();
    }
}
}

```

Мы также будем использовать базовый класс объектов-значений, о котором рассказывалось в главе 15 «Объекты-значения». Создайте в папке `Infrastructure` класс `ValueObject`, представленный в листинге 21.32.

Листинг 21.32. Базовый класс объектов-значений

```

using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class ValueObject<T> where T : ValueObject<T>
    {
        protected abstract IEnumerable<object>

```

```

GetAttributesToIncludeInEqualityCheck();

public override bool Equals(object other)
{
    return Equals(other as T);
}

public bool Equals(T other)
{
    if (other == null)
    {
        return false;
    }
    return GetAttributesToIncludeInEqualityCheck().SequenceEqual(
        other.GetAttributesToIncludeInEqualityCheck());
}

public static bool operator ==(ValueObject<T> left,
                               ValueObject<T> right)
{
    return Equals(left, right);
}

public static bool operator !=(ValueObject<T> left,
                               ValueObject<T> right)
{
    return !(left == right);
}

public override int GetHashCode()
{
    int hash = 17;
    foreach (var obj in this.GetAttributesToIncludeInEqualityCheck())
        hash = hash * 31 + (obj == null ? 0 : obj.GetHashCode());

    return hash;
}
}
}

```

Последний элемент инфраструктуры — базовый класс сущностей, представленный в листинге 21.33.

Листинг 21.33. Базовый класс сущностей

```

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }
    }
}

```

После подготовки инфраструктуры можно приступать к строительству предметной модели. В папке `Model`, в проекте `DDDDPPP.Chap21.NHibernateExample.Application`, создайте следующие две папки, где будут храниться два агрегата, составляющие предметную модель:

○ `Auction`

○ `BidHistory`

Сначала создадим агрегат `Auction`. Этот агрегат реализует все предметные правила, связанные с размещением ставок. Нам понадобится объект-значение, отображающий в модели понятие «деньги». Используя листинг 21.34, создайте объект-значение с именем `Money` в папке `Auction`.

Листинг 21.34. Объект-значение `Money`

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Money : ValueObject<Money>, IComparable<Money>
    {
        protected decimal Value { get; set; }

        public Money() : this(0m)
        {
        }

        public Money(decimal value)
        {
            ThrowExceptionIfNotValid(value);

            Value = value;
        }

        private void ThrowExceptionIfNotValid(decimal value)
        {
            if (value % 0.01m != 0)
                throw new MoreThanTwoDecimalPlacesInMoneyValueException();

            if (value < 0)
                throw new MoneyCannotBeANegativeValueException();
        }

        public Money Add(Money money)
        {
            return new Money(Value + money.Value);
        }

        public bool IsGreaterThan(Money money)
        {
            return this.Value > money.Value;
        }
    }
}
```

```

public bool IsGreaterThanOrEqualTo(Money money)
{
    return this.Value > money.Value || this.Equals(money);
}

public bool IsLessThanOrEqualTo(Money money)
{
    return this.Value < money.Value || this.Equals(money);
}

public override string ToString()
{
    return string.Format("{0}", Value);
}

public int CompareTo(Money other)
{
    return this.Value.CompareTo(other.Value);
}

protected override IEnumerable<object>
    GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() {Value};
}
}
}

```

Как рассказывалось в главе 15, объекты-значения являются неизменяемыми; программа не может изменять их состояние. Именно поэтому метод `Add` возвращает новый объект `Money` вместо изменения состояния текущего.

Конструктор включает логику, требующую, чтобы денежная сумма выражалась неотрицательным числом. В противном случае возбуждается одно из двух исключений, объявленных в листинге 21.35. Добавьте эти два класса в папку `Auction`.

Листинг 21.35. Предметные исключения

```

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class MoneyCannotBeANegativeValueException : Exception
    {
    }
}

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class MoreThanTwoDecimalPlacesInMoneyValueException : Exception
    {
    }
}

```

Следующее понятие, имеющееся в предметной области аукциона, — это понятие предложения. *Предложение* (offer) определяет максимальную сумму, которую участник торгов готов заплатить за лот на аукционе. Причина определения понятия предложения состоит в том, что только предложение, превышающее ставку с приращением, становится следующей ставкой. Класс Offer также является объектом-значением. Создайте класс Offer в папке Auction, определяемый листингом 21.36.

Листинг 21.36. Объект-значение Offer

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Offer : ValueObject<Offer>
    {
        public Offer(Guid bidderId, Money maximumBid, DateTime timeOfOffer)
        {
            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("BidderId cannot be null");

            if (maximumBid == null)
                throw new ArgumentNullException("MaximumBid cannot be null");

            if (timeOfOffer == DateTime.MinValue)
                throw new ArgumentNullException(
                    "Time of Offer must have a value");

            Bidder = bidderId;
            MaximumBid = maximumBid;
            TimeOfOffer = timeOfOffer;
        }

        public Guid Bidder { get; private set; }
        public Money MaximumBid { get; private set; }
        public DateTime TimeOfOffer { get; private set; }

        protected override IEnumerable<object>
            GetAttributesToIncludeInEqualityCheck()
        {
            return new List<Object>()
            {
                Bidder, MaximumBid, TimeOfOffer
            };
        }
    }
}
```

Аукционная цена — это текущая сумма победившей ставки; цена имеет метод, определяющий шаг приращения ставки. И снова, цена является объектом-значением. Добавьте в папку `Model` класс `Price`, определяемый листингом 21.37. Настоящая система торгов eBay имеет гораздо большее количество уровней приращения. Чтобы упростить пример, здесь оставлено только три уровня.

Листинг 21.37. Объект-значение Price

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Price : ValueObject<Price>
    {
        public Price(Money amount)
        {
            if (amount == null)
                throw new ArgumentNullException("Amount cannot be null");

            Amount = amount;
        }

        public Money Amount { get; private set; }

        public Money BidIncrement()
        {
            if (Amount.IsGreaterThanOrEqualTo(new Money(0.01m)) &&
                Amount.IsLessThanOrEqualTo(new Money(0.99m)))
                return Amount.add(new Money(0.05m));

            if (Amount.IsGreaterThanOrEqualTo(new Money(1.00m)) &&
                Amount.IsLessThanOrEqualTo(new Money(4.99m)))
                return Amount.add(new Money(0.20m));

            if (Amount.IsGreaterThanOrEqualTo(new Money(5.00m)) &&
                Amount.IsLessThanOrEqualTo(new Money(14.99m)))
                return Amount.add(new Money(0.50m));

            return Amount.add(new Money(1.00m));
        }

        public bool CanBeExceededBy(Money offer)
        {
            return offer.IsGreaterThanOrEqualTo(BidIncrement());
        }

        protected override IEnumerable<object>
            GetAttributesToIncludeInEqualityCheck()
        {
            return new List<Object>() {Amount};
        }
    }
}
```


WinningBid, еще один объект-значение, представляет собой принятое предложение участника торгов. Добавьте в папку Model класс WinningBid, определяемый листингом 21.38.

Листинг 21.38. Объект-значение WinningBid

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
    {
        public WinningBid(Guid bidder, Money maximumBid,
            Money bid, DateTime timeOfBid)
        {
            if (bidder == Guid.Empty)
                throw new ArgumentNullException("Bidder cannot be null");

            if (maximumBid == null)
                throw new ArgumentNullException("MaximumBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            Bidder = bidder;
            MaximumBid = maximumBid;
            TimeOfBid = timeOfBid;
            CurrentAuctionPrice = new Price(bid);
        }

        public Guid Bidder { get; private set; }
        public Money MaximumBid { get; private set; }
        public DateTime TimeOfBid { get; private set; }
        public Price CurrentAuctionPrice { get; private set; }

        public WinningBid RaiseMaximumBidTo(Money newAmount)
        {
            if (newAmount.IsGreaterThan(MaximumBid))
                return new WinningBid(Bidder, newAmount,
                    CurrentAuctionPrice.Amount,
                    DateTime.Now);
            else
                throw new ApplicationException(
                    "Maximum bid increase must be larger than current maximum bid.");
        }

        public bool WasMadeBy(Guid bidder)
        {
            return Bidder.Equals(bidder);
        }
    }
}
```

```

public bool CanBeExceededBy(Money offer)
{
    return CurrentAuctionPrice.CanBeExceededBy(offer);
}

public bool HasNotReachedMaximumBid()
{
    return MaximumBid.IsGreaterThan(CurrentAuctionPrice.Amount);
}

protected override IEnumerable<object>
    GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() { Bidder, MaximumBid,
                                TimeOfBid, CurrentAuctionPrice };
}
}
}

```

Предметная служба `AutomaticBidder` делает ставку от имени участника торгов, повышая ее, если прежняя его ставка была перебита. Сумма ставки выбирается такой, чтобы только остаться наивысшей ставкой, в пределах максимальной суммы, предложенной участником. Создайте в папке `Model` класс `AutomaticBidder`, определяемый листингом 21.39.

Листинг 21.39. Предметная служба `AutomaticBidder`

```

using System.Collections.Generic;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class AutomaticBidder
    {
        public IEnumerable<WinningBid> GenerateNextSequenceOfBidsAfter(
            Offer offer, WinningBid currentWinningBid)
        {
            var bids = new List<WinningBid>();

            if (currentWinningBid.MaximumBid
                .IsGreaterThanOrEqualTo(offer.MaximumBid))
            {
                var bidFromOffer = new WinningBid(offer.Bidder,
                                                    offer.MaximumBid,
                                                    offer.MaximumBid,
                                                    offer.TimeOfOffer);

                bids.Add(bidFromOffer);

                bids.Add(CalculateNextBid(bidFromOffer,
                                          new Offer(currentWinningBid.Bidder,
                                                    currentWinningBid.MaximumBid,
                                                    currentWinningBid.TimeOfBid)));
            }
        }
    }
}

```

```

    }
    else
    {
        if (currentWinningBid.HasNotReachedMaximumBid())
        {
            var currentBiddersLastBid =
                new WinningBid(currentWinningBid.Bidder,
                               currentWinningBid.MaximumBid,
                               currentWinningBid.MaximumBid,
                               currentWinningBid.TimeOfBid);

            bids.Add(currentBiddersLastBid);
            bids.Add(CalculateNextBid(currentBiddersLastBid, offer));
        }
        else
            bids.Add(new WinningBid(offer.Bidder,
                                    currentWinningBid.CurrentAuctionPrice.BidIncrement(),
                                    offer.MaximumBid, offer.TimeOfOffer));
    }
    return bids;
}

private WinningBid CalculateNextBid(WinningBid winningbid, Offer offer)
{
    WinningBid bid;

    if (winningbid.CanBeExceededBy(offer.MaximumBid))
        bid = new WinningBid(offer.Bidder, offer.MaximumBid,
                              winningbid.CurrentAuctionPrice.BidIncrement(),
                              offer.TimeOfOffer);
    else
        bid = new WinningBid(offer.Bidder, offer.MaximumBid,
                              offer.MaximumBid, offer.TimeOfOffer);

    return bid;
}
}
}
}

```

BidPlaced — предметное событие, сообщающее о размещении ставки. Добавьте его в папку `Model`, как показано в листинге 21.40.

Листинг 21.40. Объект-значение BidPlace

```

using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class BidPlaced
    {
        public BidPlaced(Guid auctionId, Guid bidderId,
                        Money amountBid, DateTime timeOfBid)
        {

```

```

        if (auctionId == Guid.Empty)
            throw new ArgumentNullException("Auction Id cannot be null");

        if (bidderId == Guid.Empty)
            throw new ArgumentNullException("Bidder Id cannot be null");

        if (amountBid == null)
            throw new ArgumentNullException("AmountBid cannot be null");

        if (timeOfBid == DateTime.MinValue)
            throw new ArgumentNullException("TimeOfBid must have a value");

        AuctionId = auctionId;
        Bidder = bidderId;
        AmountBid = amountBid;
        TimeOfMemberBid = timeOfBid;
    }

    public Guid AuctionId { get; private set; }
    public Guid Bidder { get; private set; }
    public Money AmountBid { get; private set; }
    public DateTime TimeOfMemberBid { get; private set; }
}
}

```

Класс `OutBid` в листинге 21.41 — еще одно предметное событие. Оно возникает, если ставка побита.

Листинг 21.41. Объект-значение `Outbid`

```

using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class OutBid
    {
        public OutBid(Guid auctionId, Guid bidderId)
        {
            if (auctionId == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("Bidder Id cannot be null");

            AuctionId = auctionId;
            Bidder = bidderId;
        }

        public Guid AuctionId { get; private set; }
        public Guid Bidder { get; private set; }
    }
}

```

Класс `Auction` в листинге 21.42 — это сущность и корень агрегата. Он не имеет общедоступных свойств и экспортирует единственный метод, позволяющий сделать ставку. В результате размещения новой ставки возбуждаются предметные события.

Листинг 21.42. Сущность `Auction`

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        public Auction(Guid id, Money startingPrice, DateTime endsAt)
        {
            if (id == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (startingPrice == null)
                throw new ArgumentNullException(
                    "Starting Price cannot be null");

            if (endsAt == DateTime.MinValue)
                throw new ArgumentNullException("EndsAt must have a value");

            Id = id;
            StartingPrice = startingPrice;
            EndsAt = endsAt;
        }

        private Money StartingPrice { get; set; }
        private WinningBid WinningBid { get; set; }
        private DateTime EndsAt { get; set; }

        private bool StillInProgress(DateTime currentTime)
        {
            return (EndsAt > currentTime);
        }

        public void PlaceBidFor(Offer offer, DateTime currentTime)
        {
            if (StillInProgress(currentTime))
            {
                if (FirstOffer())
                    PlaceABidForTheFirst(offer);
                else if (BidderIsIncreasingMaximumBidToNew(offer))
                    WinningBid =
                        WinningBid.RaiseMaximumBidTo(offer.MaximumBid);
                else if (WinningBid.CanBeExceededBy(offer.MaximumBid))
                {

```

```

        var newBids = new AutomaticBidder()
            .GenerateNextSequenceOfBidsAfter(offer, WinningBid);

        foreach (var bid in newBids)
            Place(bid);
    }
}

private bool BidderIsIncreasingMaximumBidToNew(Offer offer)
{
    return WinningBid.WasMadeBy(offer.Bidder) && offer.MaximumBid
        .IsGreaterThan(WinningBid.MaximumBid);
}

private bool FirstOffer()
{
    return WinningBid == null;
}

private void PlaceABidForTheFirst(Offer offer)
{
    if (offer.MaximumBid.IsGreaterThanOrEqualTo(StartingPrice))
        Place(new WinningBid(offer.Bidder, offer.MaximumBid,
            StartingPrice, offer.TimeOfOffer));
}

private void Place(WinningBid newBid)
{
    if (!FirstOffer() && WinningBid.WasMadeBy(newBid.Bidder))
        DomainEvents.Raise(new OutBid(Id, WinningBid.Bidder));

    WinningBid = newBid;
    DomainEvents.Raise(new BidPlaced(Id, newBid.Bidder,
        newBid.CurrentAuctionPrice.Amount,
        newBid.TimeOfBid));
}
}
}

```

Обратите внимание, что экземпляр класса `AutomaticBidder` создается непосредственно в классе `Auction`, а не внедряется как зависимость. Шаблон внедрения зависимостей удобно использовать, когда на выбор имеется несколько реализаций требуемого интерфейса, но так как в данном случае существует только одна реализация `AutomaticBidder` и нет необходимости в использовании подставных или фиктивных предметных служб, можно без опаски создать экземпляр напрямую.

Контракт репозитория для агрегата `Auction` имеет всего два метода: один находит аукцион по идентификатору, а другой добавляет новый аукцион. Фреймворк `NHibernate` способен автоматически отслеживать изменения, поэтому здесь репозиторий действует как интерфейс коллекции; нет никакой необходимости добав-

лять в него явный метод сохранения. Добавьте определение контракта из листинга 21.43 в папку Auction. А его реализация будет добавлена в папку Infrastructure.

Листинг 21.43. Интерфейс репозитория аукциона

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public interface IAuctionRepository
    {
        void Add(Auction auction);
        Auction FindBy(Guid Id);
    }
}
```

Второй — и последний — агрегат в действительности является простой коллекцией объектов-значений, представляющих хронологию торгов. Используя листинг 21.44, создайте новый объект-значение Bid в папке BidHistory.

Листинг 21.44. Объект-значение Bid

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
{
    public class Bid : ValueObject<Bid>
    {
        public Bid(Guid auctionId, Guid bidderId, Money amountBid,
            DateTime timeOfBid)
        {
            if (auctionId == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("Bidder Id cannot be null");

            if (amountBid == null)
                throw new ArgumentNullException("AmountBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            AuctionId = auctionId;
            Bidder = bidderId;
            AmountBid = amountBid;
            TimeOfBid = timeOfBid;
        }

        public Guid AuctionId { get; private set; }
```

```

public Guid Bidder { get; private set; }
public Money AmountBid {get; private set;}
public DateTime TimeOfBid { get; private set; }

protected override IEnumerable<object>
    GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() { Bidder, AuctionId, TimeOfBid, AmountBid };
}
}
}

```

Другой единственный компонент этого агрегата — контракт репозитория, представленный в листинге 21.45. Репозиторий экспортирует два метода, один из которых возвращает ставки, размещенные на указанном аукционе, а другой сохраняет новую ставку.

Листинг 21.45. Интерфейс репозитория хронологии ставок

```

using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid auctionId);
        void Add(Bid bid);
    }
}

```

На этом создание предметной модели приложения закончено. Получившееся решение должно выглядеть, как показано на рис. 21.11.

Прикладные службы

Завершив создание предметной модели, можно приступить к конструированию уровня прикладных служб, которые будут действовать как клиенты предметной модели. Добавьте в папку `Application` следующие две папки:

- `BusinessUseCases`
- `Queries`

Папка `BusinessUseCases` будет содержать все функции и сценарии использования приложения, а папка `Queries` — все запросы, необходимые для составления отчетов о состоянии предметной модели.

Чтобы создать новый аукцион, необходимо определить начальную цену и конечную дату. Передача информации, необходимой для создания аукциона, будет осуществляться клиентами прикладной службы посредством объекта переноса данных (`Data Transfer Object`, `DTO`). Используя листинг 21.46, создайте `DTO` в папке `BusinessUseCases`.

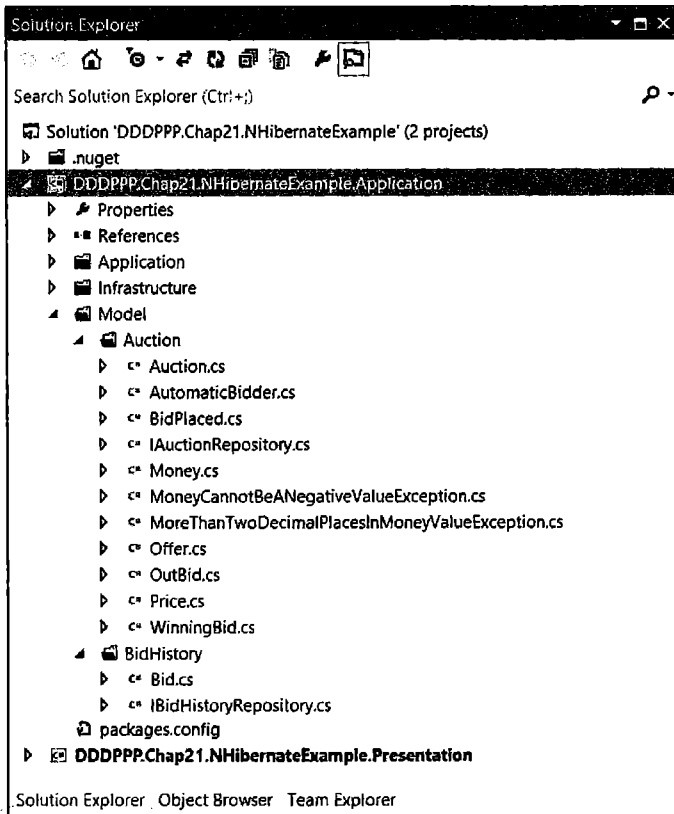


Рис. 21.11. Структура решения в Visual Studio

Листинг 21.46. Команда создания нового аукциона

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
    public class NewAuctionRequest
    {
        public decimal StartingPrice { get; set; }
        public DateTime EndsAt { get; set; }
    }
}
```

Реализация прикладной службы, обрабатывающей эту команду, представлена в листинге 21.47. Прикладная служба `CreateAuction` просто создает новый аукцион и добавляет его в репозиторий. Отметьте, что операция завернута в транзакцию `NHibernate`. Прикладная служба получает реализацию шаблона «Единица работы» из `NHibernate` через параметр конструктора и сохраняет в переменной `ISession`. Как будет показано далее, в конструктор передается тот же экземпляр

`ISession`, что используется в репозитории, в результате образуется связь между единицей работы и репозиторием в рамках прикладной службы. Эта связь гарантирует, что управление подтверждением изменений предметных объектов останется на уровне прикладных служб.

Листинг 21.47. Прикладная служба создания аукциона

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using NHibernate;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctionRepository;
        private ISession _unitOfWork;

        public CreateAuction(IAuctionRepository auctionRepository,
                             ISession unitOfWork)
        {
            _auctionRepository = auctionRepository;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            using (ITransaction transaction = _unitOfWork.BeginTransaction())
            {
                _auctionRepository.Add(new Auction(auctionId,
                                                    startingPrice, command.EndsAt));

                transaction.Commit();
            }
            return auctionId;
        }
    }
}
```

`ISession` — это основной интерфейс, обеспечивающий сохранение и извлечение предметных сущностей, который может рассматриваться как шлюз в `NHibernate` к базе данных. Документация с описанием `NHibernate` определяет `ISession` как «диспетчер хранилища». `ISession` — это реализация единицы работы в `NHibernate`. Поскольку интерфейс `ISession` реализует шаблон «Единица работы», обсуждавшийся выше в этой главе, никакие изменения не будут сохранены, пока не произойдет подтверждение транзакции. Во фреймворке `NHibernate` реализован еще один шаблон — шаблон «Карта соответствия» (`Identity map`), который гарантиру-

ет присутствие единственного экземпляра предметной сущности в `ISession` независимо от количества попыток извлечь ее.

Время — важное понятие в предметной области аукциона, и ставки должны приниматься, только пока аукцион остается активным. Обычно связывать предметные объекты с системными часами считается нецелесообразным, потому что это усложняет тестирование. Имея это в виду, абстрагируем понятие часов, определив интерфейс `IClock` в папке `Infrastructure`, как показано в листинге 21.48.

Листинг 21.48. Интерфейс для класса часов

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public interface IClock
    {
        DateTime Time();
    }
}
```

Для прикладных нужд будут использоваться системные часы. Создайте реализацию интерфейса `IClock` в папке `Infrastructure`, используя листинг 21.49.

Листинг 21.49. Реализация интерфейса часов

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public class SystemClock : IClock
    {
        public DateTime Time()
        {
            return DateTime.Now;
        }
    }
}
```

Второй сценарий использования, для реализации которого будет создана еще одна прикладная служба, — это предложение ставки на аукционе. Используя листинг 21.50, создайте класс `BidOnAuction`.

Листинг 21.50. Команда предложения ставки на аукционе

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using NHibernate;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
```

```
public class BidOnAuction
{
    private IAuctionRepository _auctionRepository;
    private IBidHistoryRepository _bidHistoryRepository;
    private ISession _unitOfWork;
    private IClock _clock;

    public BidOnAuction(IAuctionRepository auctionRepository,
                        IBidHistoryRepository bidHistoryRepository,
                        ISession unitOfWork, IClock clock)
    {
        _auctionRepository = auctionRepository;
        _bidHistoryRepository = bidHistoryRepository;
        _unitOfWork = unitOfWork;
        _clock = clock;
    }

    public void Bid(Guid auctionId, Guid memberId, decimal amount)
    {
        try
        {
            using (ITransaction transaction =
                    _unitOfWork.BeginTransaction())
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctionRepository.FindBy(auctionId);
                    var bidAmount = new Money(amount);
                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                                _clock.Time(), _clock.Time()));
                }
                transaction.Commit();
            }
        }
        catch (StaleObjectStateException ex)
        {
            _unitOfWork.Clear();

            Bid(auctionId, memberId, amount);
        }
    }

    private Action<BidPlaced> BidPlaced()
    {
        return (BidPlaced e) =>
        {
            var bidEvent = new Bid(e.AuctionId, e.Bidder, e.AmountBid,
                                    e.TimeOfMemberBid);

            _bidHistoryRepository.Add(bidEvent);
        };
    }
}
```

```

    }

    private Action<OutBid> OutBid()
    {
        return (OutBid e) =>
        {
            // Послать электронное письмо участнику с сообщением,
            // что его ставка перебита
        };
    }
}
}

```

И снова операция завернута в транзакцию NHibernate. Это важно, потому что успешное размещение ставки возбуждает предметное событие, которое приводит к добавлению ставки в `BidHistoryRepository`. Тело метода, размещающего ставку, также заключено в конструкцию `try...catch`, перехватывающую исключение `StaleObjectStateException`, при возникновении которого выполняется повторная попытка разместить ставку. Исключение `StaleObjectStateException` возникает, если аукцион был изменен другим участником в период между его извлечением из репозитория и подтверждением единицы работы. В этом примере производится повторная попытка разместить ставку в обновленной, последней версии аукциона.

Реализация репозитория NHibernate

Закончив создание модели и уровня прикладных служб, сфокусируемся на реализации репозитория NHibernate. Первое, что нужно сделать, — определить отображение агрегатов. Используя листинг 21.51, добавьте новый XML-файл с именем `Auction.hbm.xml` в папку `Infrastructure`.

Листинг 21.51. XML-отображение класса Auction для NHibernate

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
    namespace="DDDDPPP.Chap21.NHibernateExample.Application.Model.Auction"
    assembly="DDDDPPP.Chap21.NHibernateExample.Application">

    <class name="Auction" table="Auctions" lazy="false" >

        <id name="Id" column="Id" type="Guid">
            </id>

        <version name="Version" column="Version" type="Int32" unsaved-value="0"/>

        <component name="StartingPrice" class="Money">
            <property name="Value" column="StartingPrice" not-null="true"/>
        </component>

        <property name="EndsAt" column="AuctionEnds" not-null="true"/>

        <component name="WinningBid" class="WinningBid">
            <property name="Bidder" column="BidderMemberId" not-null="false"/>

            <property name="TimeOfBid" column="TimeOfBid" not-null="false"/>
        </component>
    </class>
</hibernate-mapping>

```

```

<component name="MaximumBid" class="Money">
  <property name="Value" column="MaximumBid" not-null="false"/>
</component>

<component name="CurrentAuctionPrice" class="Price">
  <component name="Amount" class="Money">
    <property name="Value" column="CurrentPrice" not-null="false"/>
  </component>
</component>
</component>
</class>
</hibernate-mapping>

```

В этой главе не обсуждается синтаксис таких файлов, потому что тема использования NHibernate не является предметом данной книги, но пример достаточно очевидно демонстрирует, как NHibernate отображает предметные сущности и их свойства в таблицы и столбцы. Более детальные сведения об NHibernate можно найти во множестве ресурсов в Интернете или в книге «NHibernate in Action».

Чтобы включить в работу файл отображения для NHibernate, установите в поле **Build Action** (Действие при сборке) значение **Embedded Resource** (Встроенный ресурс), как показано на рис. 21.12.

Для отображения ставки добавьте в папку **Infrastructure** второй XML-файл с именем **Bid.hbm.xml**, с содержимым из листинга 21.52. Точно так же не забудьте для этого файла установить в поле **Build Action** (Действие при сборке) значение **Embedded Resource** (Встроенный ресурс).

Листинг 21.52. XML-отображение класса хронологии ставок для NHibernate

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
  namespace="DDDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory"
  assembly="DDDDPPP.Chap21.NHibernateExample.Application">

  <class name="Bid" table="BidHistory" lazy="false" >

    <id name="Id" column="Id" type="Guid">
      <generator class="guid"/>
    </id>

    <property name="AuctionId" column="AuctionId" not-null="false"/>
    <property name="Bidder" column="BidderId" not-null="false"/>

    <component name="AmountBid"
      class=
        "DDDDPPP.Chap21.NHibernateExample.Application.Model.Auction.Money">
      <property name="Value" column="Bid" not-null="false"/>
    </component>

    <property name="TimeOfBid" column="TimeOfBid" not-null="false"/>
  </class>
</hibernate-mapping>

```

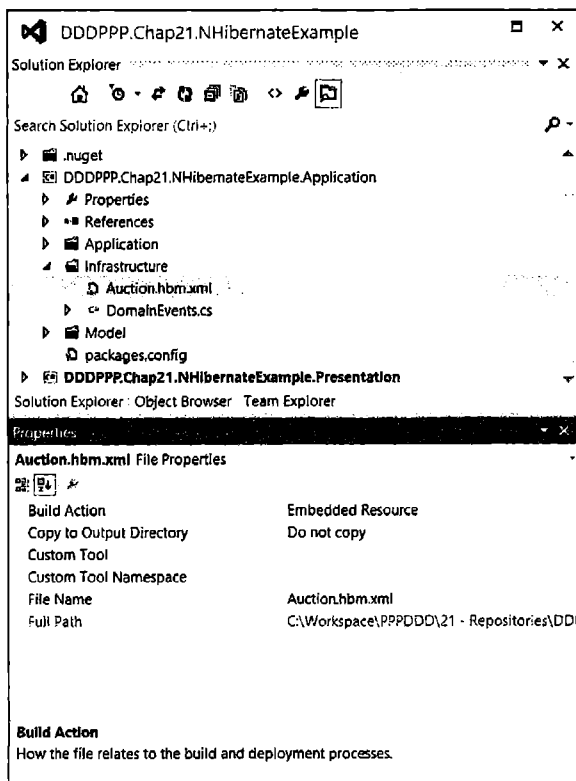


Рис. 21.12. Свойство Build Action (Действие при сборке) XML-файла отображения

Реализации репозитория находятся в пространстве имен инфраструктуры. Первая из них — реализация репозитория аукциона — определена в листинге 21.53.

Листинг 21.53. Реализация репозитория аукциона

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly ISession _session;

        public AuctionRepository(ISession session)
        {
            _session = session;
        }
    }
}
```

```

    }

    public void Add(Auction auction)
    {
        _session.Save(auction);
    }

    public Auction FindBy(Guid Id)
    {
        return _session.Get<Auction>(Id);
    }
}
}

```

Как можно видеть, переменная `ISession` выполняет всю основную работу, и именно поэтому так важно, чтобы через конструктор внедрялся тот же экземпляр, что внедряется в прикладные службы через их конструкторы. В противном случае прикладные службы не смогли бы управлять единицей работы.

Реализация репозитория `BidHistory`, представленная в листинге 21.54, также проста — с единственным отличием, заключающимся в дополнительном запросе, возвращающем сводную информацию о числе ставок в аукционе.

Листинг 21.54. Реализация репозитория хронологии ставок

```

using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using NHibernate;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
        private readonly ISession _session;

        public BidHistoryRepository(ISession session)
        {
            _session = session;
        }

        public int NoOfBidsFor(Guid auctionId)
        {
            var sql = String.Format(
                "SELECT Count(*) FROM BidHistory Where AuctionId = '{0}'",
                auctionId);

            var query = _session.CreateSQLQuery(sql);
            var result = query.UniqueResult();

            return Convert.ToInt32(result);
        }
    }
}

```



```

        public void Add(BidEvent bid)
        {
            _session.Save(bid);
        }
    }
}

```

Фреймворк NHibernate имеет несколько ограничений. Одно из них требует, чтобы все хранимые объекты имели конструктор без параметров. К счастью, конструктор может быть приватным, поэтому данное ограничение не оказывает существенного влияния на модель. Это малая цена, требуемая фреймворком, за возможность определить предметную модель как РОО.

Дополните класс Auction, добавив в него приватный конструктор, как показано в листинге 21.55.

Листинг 21.55. Сущность Auction

```

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        private Auction() { }

        public Auction(Guid id, Money startingPrice, DateTime endsAt)
        {
            Id = id;
            StartingPrice = startingPrice;
            EndsAt = endsAt;
        }

        // .....
    }
}

```

Дополните класс WinningBid, добавив в него приватный конструктор, как показано в листинге 21.56.

Листинг 21.56. Объект-значение WinningBid

```

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
    {
        private WinningBid() { }

        public WinningBid(Guid bidder, Money maximumBid, Money bid, DateTime
            timeOfBid)
        {
            //.....
        }
    }
}

```

Дополните класс Price, добавив в него приватный конструктор, как показано в листинге 21.57.

Листинг 21.57. Объект-значение Price

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Price : ValueObject<Price>
    {
        private Price() { }

        public Price(Money amount)
        {
            //.....
        }
    }
}
```

Обратите внимание на присутствие свойства `Id` в определении отображения для класса `Bid`. NHibernate требует, чтобы все отображаемые объекты имели свойство-идентификатор. Дополните класс `Bid`, добавив приватный конструктор и новое свойство для хранения идентификатора, которое будет устанавливаться фреймворком NHibernate, как показано в листинге 21.58.

Листинг 21.58. Объект-значение Bid

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
{
    public class Bid : ValueObject<Bid>
    {
        private Guid Id { get; set; }

        private Bid()
        { }

        // .....
    }
}
```

Отметьте также, что определение отображения для класса `Auction` включает свойство `Version`. Это свойство помогает определить факт изменения данных с момента последнего извлечения их из базы данных. Когда NHibernate сохраняет сущность, он проверяет совпадение версий; если они не равны — возбуждается исключение `StaleObjectStateException`. Данное исключение обрабатывается прикладной службой и реализацией предметной модели аукциона; если это произошло, выполняется повторная попытка сделать предложение.

Добавьте свойство для хранения версии в класс `Entity`, как показано в листинге 21.59.

Листинг 21.59. Базовый класс Entity

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }

        public int Version { get; private set; }
    }
}
```

Схема базы данных

В качестве базы данных можно использовать бесплатную версию MS SQL Express (<http://www.microsoft.com/web/platform/database.aspx>). Схему базы данных можно получить из отображения; в действительности фреймворк NHibernate способен сконструировать базу данных, опираясь на отображение. Тем не менее ниже приводится полное определение схемы. Создайте базу данных с именем AuctionExample и выполните сценарий из листинга 21.60, чтобы настроить ее.

Листинг 21.60. Схема базы данных

```
USE [AuctionExample]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[BidHistory](
[AuctionId] [uniqueidentifier] NOT NULL,
[BidderId] [uniqueidentifier] NOT NULL,
[Bid] [numeric](18, 2) NOT NULL,
[TimeOfBid] [datetime] NOT NULL,
[Id] [uniqueidentifier] NOT NULL,
CONSTRAINT [PK_BidHistory] PRIMARY KEY CLUSTERED
(
[Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Auctions](
[Id] [uniqueidentifier] NOT NULL,
[StartingPrice] [decimal](18, 2) NOT NULL,
[BidderMemberId] [uniqueidentifier] NULL,
[TimeOfBid] [datetime] NULL,
[MaximumBid] [decimal](18, 2) NULL,
[CurrentPrice] [decimal](18, 2) NULL,
[AuctionEnds] [datetime] NOT NULL,
[Version] [int] NOT NULL,
CONSTRAINT [PK_Auctions] PRIMARY KEY CLUSTERED
(
[Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Получившаяся база данных должна иметь структуру, изображенную на рис. 21.13.

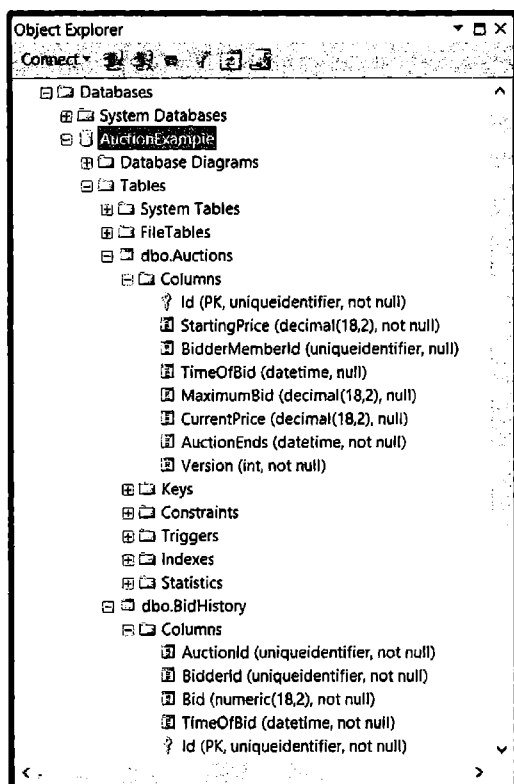


Рис. 21.13. Схема базы данных

Справочные службы

Обратимся теперь к реализации справочных служб в уровне прикладных служб. Справочные службы сообщают информацию об аукционах. Первое представление содержит сводную информацию о состоянии аукциона. Вместо предметного объекта мы будем использовать простую модель представления. Используя листинг 21.61, создайте класс `AuctionStatus` в папке `Queries`.

Листинг 21.61. Класс состояния аукциона

```
using System;
namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class AuctionStatus
    {
        public Guid Id { get; set; }
        public decimal CurrentPrice { get; set; }
        public DateTime AuctionEnds { get; set; }
    }
}
```

```

    public Guid WinningBidderId { get; set; }
    public int NumberOfBids { get; set; }
    public TimeSpan TimeRemaining {get; set;}
}
}

```

Для получения состояния аукциона необходимо использовать SQL-запрос из-за отсутствия общедоступных методов чтения (getters), реализующих преобразование агрегата в DTO-объект `AuctionStatus`. Такое решение ясно отделяет предметную модель от потребностей приложения в отчетах и избавляет репозиторий от необходимости поддерживать задачи, связанные с отчетами.

Листинг 21.62. Состояние аукциона

```

using System;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class AuctionStatusQuery
    {
        private readonly ISession _session;
        private readonly IBidHistoryRepository _bidHistory;
        private readonly IClock _clock;

        public AuctionStatusQuery(ISession session,
                                IBidHistoryRepository bidHistory,
                                IClock clock)
        {
            _session = session;
            _bidHistory = bidHistory;
            _clock = clock;
        }

        public AuctionStatus AuctionStatus(Guid auctionId)
        {
            var status = _session
                .CreateSQLQuery(String.Format(
                    "select Id, CurrentPrice, BidderMemberId as WinningBidderId, " +
                    "AuctionEnds from Auctions Where Id = '{0}'", auctionId))
                .SetResultTransformer(
                    Transformers.AliasToBean<AuctionStatus>())
                .UniqueResult<AuctionStatus>();

            status.TimeRemaining = TimeRemaining(status.AuctionEnds);
            status.NumberOfBids = _bidHistory.NoOfBidsFor(auctionId);

            return status;
        }
    }
}

```

```

public TimeSpan TimeRemaining(DateTime AuctionEnds)
{
    if (_clock.Time() < AuctionEnds)
        return AuctionEnds.Subtract(_clock.Time());
    else
        return new TimeSpan();
}
}
}

```

Другой отчет, необходимый приложению, дает доступ к хронологии ставок на аукционе. Здесь также нежелательно использовать предметные объекты, поэтому создадим специализированный объект DTO, представленный в листинге 21.63. Несмотря на большое сходство с предметным объектом `Bid`, класс `BidInformation` решает совершенно иную задачу и не будет затронут в случае изменения объекта-значения `Bid`.

Листинг 21.63. Объект переноса данных с информацией о ставке

```

using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class BidInformation
    {
        public Guid Bidder { get; set; }
        public decimal AmountBid { get; set; }
        public string currency { get; set; }
        public DateTime TimeOfBid { get; set; }
    }
}

```

И опять справочная служба, представленная в листинге 21.64, которая возвращает справочную информацию о ставках, размещавшихся в аукционе, вынуждена непосредственно обращаться к базе данных из-за закрытости предметной модели.

Листинг 21.64. Служба, возвращающая хронологию ставок

```

using System;
using System.Collections.Generic;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly ISession _session;

        public BidHistoryQuery(ISession session)
        {
            _session = session;
        }
    }
}

```

```

public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
{
    var status = _session
        .CreateSQLQuery(String.Format(
            "SELECT [BidderId] as Bidder,[Bid] as AmountBid ,TimeOfBid " +
            "FROM [BidHistory] " +
            "Where AuctionId = '{0}' "+
            "Order By Bid Desc, TimeOfBid Asc", auctionId))
        .SetResultTransformer(
            Transformers.AliasToBean<BidInformation>());

    return status.List<BidInformation>();
}
}
}

```

Конфигурация

Для поддержки фреймворка необходимо определить некоторые настройки. Измените файл `app.config` в проекте `Presentation`, чтобы он соответствовал содержанию листинга 21.65. Затем измените строку подключения в зависимости от имени экземпляра вашей базы данных.

Листинг 21.65. Конфигурационный файл для NHibernate

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <configSections>
        <section name="hibernate-configuration"
            type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
    </configSections>

    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>

    <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
        <session-factory name="NHibernate.Test">
            <property name="connection.driver_class">
                NHibernate.Driver.SqlClientDriver</property>
            <property name="connection.connection_string">
                Data Source=.\SQLEXPRESS;Database=AuctionExample;Trusted_Connection=True;
            </property>
            <property name="adonet.batch_size">10</property>
            <property name="show_sql">false</property>
            <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
            <property name="command_timeout">60</property>
            <property name="query.substitutions">true 1, false 0, yes 'Y', no 'N'
            </property>
        </session-factory>
    </hibernate-configuration>
</configuration>

```


Обычно объект `ISessionFactory` создается в единственном экземпляре из-за довольно высокой стоимости операции его создания. Одной из обязанностей `ISessionFactory` является создание экземпляров `ISession`. Здесь `StructureMap` гарантирует, что всеми репозиториями и прикладными службами будет использоваться одна и та же версия переменной `ISession`, а это значит, что прикладные службы смогут контролировать единицы работы.

Представление

Наконец, чтобы показать, как действует получившееся приложение, симитируем размещение ставок пользователей в ходе торгов. Добавьте содержимое листинга 21.67 в файл `Program` в проекте `Presentation`.

Листинг 21.67. Программа с интерфейсом командной строки

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases;
using DDDPPP.Chap21.NHibernateExample.Application.Application.Queries;
using DDDPPP.Chap21.NHibernateExample.Application;
using StructureMap;

namespace DDDPPP.Chap21.NHibernateExample.Presentation
{
    public class Program
    {
        private static Dictionary<Guid, String> members
            = new Dictionary<Guid, string>();

        public static void Main(string[] args)
        {
            Bootstrapper.Startup();

            var memberIdA = Guid.NewGuid();
            var memberIdB = Guid.NewGuid();

            members.Add(memberIdA, "Ted");
            members.Add(memberIdB, "Rob");

            var auctionId = CreateAuction();

            Bid(auctionId, memberIdA, 10m);
            Bid(auctionId, memberIdB, 1.49m);
            Bid(auctionId, memberIdB, 10.01m);
            Bid(auctionId, memberIdB, 12.00m);
            Bid(auctionId, memberIdA, 12.00m);
        }

        public static Guid CreateAuction()
        {
            var createAuctionService =
                ObjectFactory.GetInstance<CreateAuction>();
```

```

        var newAuctionRequest = new NewAuctionRequest();

        newAuctionRequest.StartingPrice = 0.99m;
        newAuctionRequest.EndsAt = DateTime.Now.AddDays(1);

        var auctionId = createAuctionService.Create(newAuctionRequest);

        return auctionId;
    }

    public static void Bid(Guid auctionId, Guid memberId, decimal amount)
    {
        var bidOnAuctionService = ObjectFactory.GetInstance<BidOnAuction>();

        bidOnAuctionService.Bid(auctionId, memberId, amount);

        PrintStatusOfAuctionBy(auctionId);
        PrintBidHistoryOf(auctionId);

        Console.WriteLine("Hit any key to continue");
        Console.ReadLine();
    }

    public static void PrintStatusOfAuctionBy(Guid auctionId)
    {
        var auctionSummaryQuery =
            ObjectFactory.GetInstance<AuctionStatusQuery>();
        var status = auctionSummaryQuery.AuctionStatus(auctionId);

        Console.WriteLine("No Of Bids: " + status.NumberOfBids);
        Console.WriteLine("Current Bid: " +
            status.CurrentPrice.ToString("##.##"));
        Console.WriteLine("Winning Bidder: " +
            FindNameOfBidderWith(status.WinningBidderId));
        Console.WriteLine("Time Remaining: " + status.TimeRemaining);
        Console.WriteLine();
    }

    public static void PrintBidHistoryOf(Guid auctionId)
    {
        var bidHistoryQuery = ObjectFactory.GetInstance<BidHistoryQuery>();
        var status = bidHistoryQuery.BidHistoryFor(auctionId);

        Console.WriteLine("Bids..");

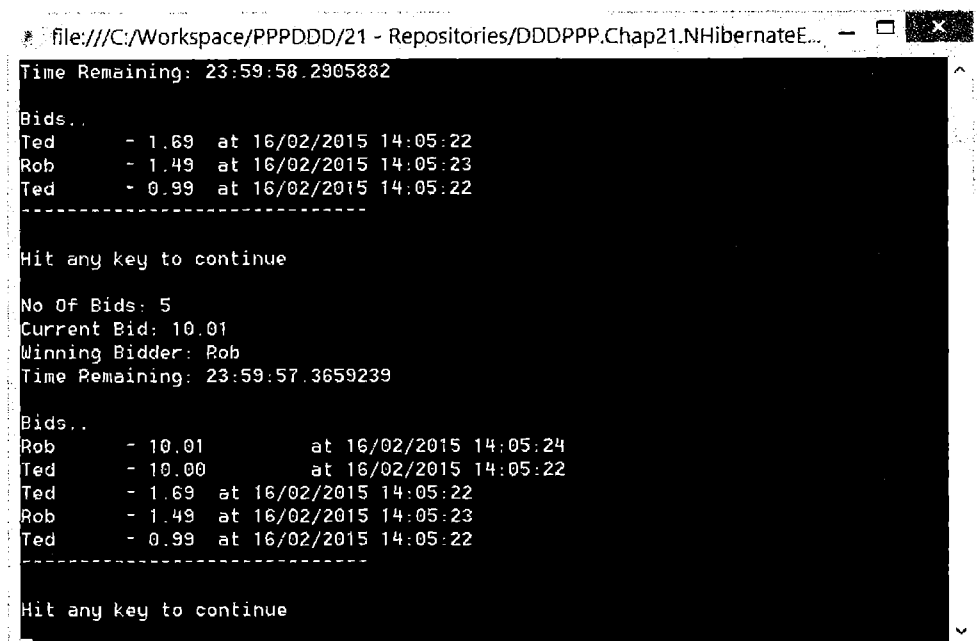
        foreach (var bid in status)
            Console.WriteLine(FindNameOfBidderWith(bid.Bidder) +
                "\t - " +
                bid.AmountBid.ToString("G") +
                "\t at " + bid.TimeOfBid);
        Console.WriteLine("-----");
    }

```

```
        Console.WriteLine();
    }

    public static string FindNameOfBidderWith(Guid id)
    {
        if (members.ContainsKey(id))
            return members[id];
        else
            return string.Empty;
    }
}
```

На рис. 21.14 изображен результат выполнения программы.



```
file:///C:/Workspace/PPPD/21 - Repositories/DDDDPP.Chap21.NHibernateE...
Time Remaining: 23:59:58.2905882

Bids..
Ted      - 1.69   at 16/02/2015 14:05:22
Rob      - 1.49   at 16/02/2015 14:05:23
Ted      - 0.99   at 16/02/2015 14:05:22
-----

Hit any key to continue

No Of Bids: 5
Current Bid: 10.01
Winning Bidder: Rob
Time Remaining: 23:59:57.3659239

Bids..
Rob      - 10.01   at 16/02/2015 14:05:24
Ted      - 10.00   at 16/02/2015 14:05:22
Ted      - 1.69   at 16/02/2015 14:05:22
Rob      - 1.49   at 16/02/2015 14:05:23
Ted      - 0.99   at 16/02/2015 14:05:22
-----

Hit any key to continue
```

Рис. 21.14. Результат выполнения программы

Пример для RavenDB

RavenDB — это документоориентированная база данных, не имеющая предопределенной схемы и хранящая предметные объекты в виде документов в формате JSON (JavaScript Object Notation — форма записи объектов JavaScript). По этой причине модель данных и предметная модель полностью совпадают друг с другом. В данном примере мы изменим предметную модель, сделав ее свойства общедоступными, однако методы записи (setters) оставим приватными.

Настройка решения

Прежде чем приступить к созданию решения в Visual Studio, необходимо установить RavenDB. Проще всего сделать это, загрузив последнюю версию установочного комплекта с веб-сайта RavenDB (<http://ravendb.net/download>). Экземпляры серверов RavenDB поддерживают возможность удаленного управления посредством приложения RavenDB Management Studio, реализованного на основе технологии Silverlight. После установки вы сможете открыть в браузере главную страницу приложения RavenDB Management Studio, доступную по адресу <http://localhost:8080/>.

После установки RavenDB создайте в Visual Studio пустое решение с именем `DDDDPPPP.Chap21.RavenDBExample` и добавьте в него библиотеку классов `DDDDPPPP.Chap21.RavenDBExample.Application`, а также консольное приложение `DDDDPPPP.Chap21.RavenDBExample.Presentation`. Добавьте в проект `Presentation` ссылку на библиотеку `Application`. Далее создайте в проекте `DDDDPPPP.Chap21.RavenDBExample.Application` следующие папки:

- Application
- Infrastructure
- Model

Файл `Class1.cs`, созданный автоматически, можно удалить, потому что он нам не понадобится. С помощью NuGet установите клиентские библиотеки RavenDB, как показано на рис. 21.15.

Создайте приложение, как это описывается в примере для NHibernate, или просто скопируйте файлы классов в новое решение, не забыв при этом изменить назва-

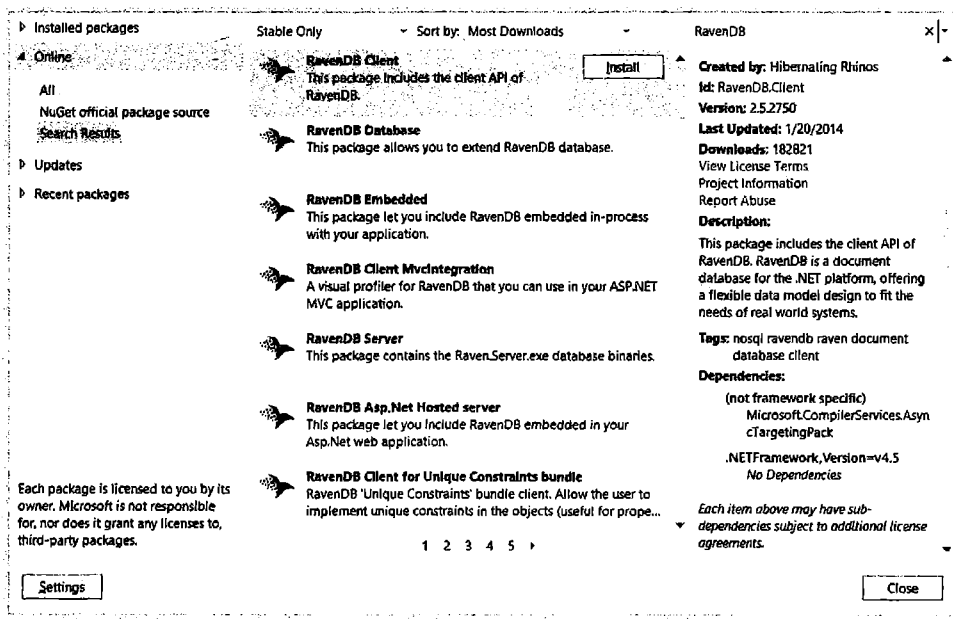


Рис. 21.15. Установка клиентских библиотек RavenDB с помощью NuGet

ния пространств имен. Удалите следующие классы, специфические для примера на основе NHibernate:

- BusinessUseCases\BidOnAuction.cs
- BusinessUseCases\CreateAuction.cs
- Queries\AuctionStatusQuery.cs
- Queries\BidHistoryQuery.cs
- Infrastructure\Auction.hbm.xml
- Infrastructure\Bid.hbm.xml
- Infrastructure\AuctionRepository.cs
- Infrastructure\BidHistoryRepository.cs
- Bootstrapper.cs

После этого решение должно выглядеть, как показано на рис. 21.16.

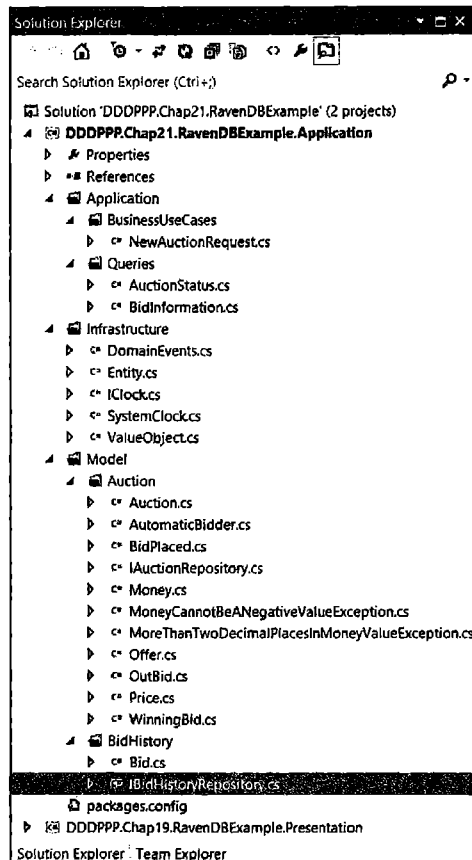


Рис. 21.16. Содержимое панели обозревателя решения с приложением, основанного на примере для NHibernate

Классы, которые пришлось удалить, заменит реализация репозитория RavenDB.

Изменение модели

Предметная модель останется почти идентичной предметной модели в примере для NHibernate. Предметные объекты все так же должны включать конструкторы без параметров. Однако так как теперь методы чтения (getters) должны быть общедоступными, измените свойства класса Auction, как показано в листинге 21.68.

Листинг 21.68. Изменение свойств в классе Auction

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        private Auction() { }

        ....

        public Money StartingPrice { get; private set; }
        public WinningBid WinningBid { get; private set; }
        public DateTime EndsAt { get; private set; }

        ...

        public bool HasBeenBidOn()
        {
            return WinningBid == null;
        }
    }
}
```

В классе Auction появился также метод HasBeenBidOn, который будет использоваться справочной службой при составлении отчета о состоянии аукциона. Еще одно изменение коснулось метода чтения свойства Value в классе Money — этот метод сделан общедоступным для нужд составления отчетов, как показано в листинге 21.69.

Листинг 21.69. Изменение свойства Value класса Money

```
namespace DDDPPP.Chap21.RavenDBExample.Application.Model.Auction
{
    public class Money
    {
        public decimal Value { get; private set; }
    }
}
```

В пример для RavenDB необходимо также включить дополнительный предметный объект — `BidHistory`. Он хранит все ставки, размещенные на аукционе, в порядке их добавления. Добавьте этот класс в папку `BidHistory` предметной модели, используя листинг 21.70.

Листинг 21.70. Класс хронологии ставок

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory
{
    public class BidHistory
    {
        private IEnumerable<Bid> _bids;

        public BidHistory(IEnumerable<Bid> bids)
        {
            if (bids == null)
                throw new ArgumentNullException("Bids cannot be null");

            _bids = bids;
        }

        public IEnumerable<Bid> ShowAllBids()
        {
            var bids = _bids.OrderByDescending(x => x.AmountBid)
                               .ThenBy(x => x.TimeOfBid);

            return bids;
        }
    }
}
```

Для извлечения объектов `BidHistory` нужно добавить новый метод в контракт `IBidHistoryRepository`, как показано в листинге 21.71.

Листинг 21.71. Новый метод в репозитории хронологии ставок

```
using System;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid auctionId);
        void Add(Bid bid);
        BidHistory FindBy(Guid auctionId);
    }
}
```

Прикладные службы

Прикладные службы — единственное, что существенно отличает данное решение от решения на основе NHibernate. Это объясняется использованием типа `IDocumentSession` из RavenDB вместо `ISession` из NHibernate. Также отличается исключение, возбуждаемое в случае появления проблем, связанных с одновременным доступом. RavenDB поддерживает неявные транзакции, встроенные в `IDocumentSession`. Благодаря этому отпала необходимость явно обертыывать операции транзакциями, как показано в листинге 21.72.

Листинг 21.72. Измененная версия класса прикладной службы BidOnAuction

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;
using Raven.Abstractions.Exceptions;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.BusinessUseCases
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctions;
        private IBidHistoryRepository _bidHistory;
        private IDocumentSession _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctions,
                           IBidHistoryRepository bidHistory,
                           IDocumentSession unitOfWork,
                           IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctions.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                                  _clock.Time(), _clock.Time()));
                }
            }
        }
    }
}
```



```

        }
        _unitOfWork.SaveChanges();
    }
    catch (ConcurrencyException ex)
    {
        _unitOfWork.Advanced.Clear();
        Bid(auctionId, memberId, amount);
    }
}

private Action<BidPlaced> BidPlaced()
{
    return (BidPlaced e) =>
    {
        var bidEvent = new Bid(e.AuctionId, e.Bidder,
                               e.AmountBid, e.TimeOfBid);
        _bidHistory.Add(bidEvent);
    };
}

private Action<OutBid> OutBid()
{
    return (OutBid e) =>
    {
        // Послать электронное письмо участнику с сообщением,
        // что его ставка перебита
    };
}
}
}

```

При использовании RavenDB нет необходимости использовать свойство `Version` в базовом классе `Entity`, потому что поддержка версий встраивается по умолчанию. Когда документ загружается из RavenDB, вместе с ним в кэше сохраняется соответствующий признак `Etag`. Признак `Etag` по своей сути является отражением текущей версии. Когда производится подтверждение изменений в сеансе, RavenDB проверяет возможное изменение признака `Etag` документа с момента его извлечения и возбуждает исключение `ConcurrencyException`, если значение `Etag` изменилось.

Прикладная служба `CreateAuction` напоминает одноименную службу из реализации на основе NHibernate, как показано в листинге 21.73.

Листинг 21.73. Прикладная служба создания аукциона в примере для RavenDB

```

using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.BusinessUseCases
{

```



```

        _auctions = auctions;
        _bidHistory = bidHistory;
        _clock = clock;
    }

    public AuctionStatus AuctionStatus(Guid auctionId)
    {
        var auction = _auctions.FindBy(auctionId);
        var status = new AuctionStatus();

        status.AuctionEnds = auction.EndsAt;
        status.Id = auction.Id;

        if (auction.HasBeenBidOn())
        {
            status.CurrentPrice =
                auction.WinningBid.CurrentAuctionPrice.Amount.Value;
            status.WinningBidderId = auction.WinningBid.Bidder;
        }

        status.TimeRemaining = TimeRemaining(auction.EndsAt);
        status.NumberOfBids = _bidHistory.NoOfBidsFor(auctionId);

        return status;
    }

    public TimeSpan TimeRemaining(DateTime AuctionEnds)
    {
        if (_clock.Time() < AuctionEnds)
            return AuctionEnds.Subtract(_clock.Time());
        else
            return new TimeSpan();
    }
}

```

Справочная служба, которая возвращает список ставок для указанного аукциона, использует объект `BidHistory`, чтобы гарантировать правильный порядок их следования, так как это относится к предметной логике. Добавьте в решение класс `BidHistoryQuery`, используя листинг 21.75.

Листинг 21.75. Класс `BidHistoryQuery`

```

using System;
using System.Collections.Generic;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.RavenDBExample.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly IBidHistoryRepository _bidHistory;
    }
}

```

```

public BidHistoryQuery(IBidHistoryRepository bidHistory)
{
    _bidHistory = bidHistory
}

public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
{
    var bidHistory = _bidHistory.FindBy(auctionId);

    return Convert(bidHistory.ShowAllBids());
}

public IEnumerable<BidInformation> Convert(IEnumerable<Bid> bids)
{
    var bidInfo = new List<BidInformation>();

    foreach (var bid in bids)
    {
        bidInfo.Add(new BidInformation() { Bidder = bid.Bidder,
                                           AmountBid = bid.AmountBid.Value,
                                           TimeOfBid = bid.TimeOfBid });
    }

    return bidInfo;
}
}
}

```

Реализация репозитория

Как вы помните, контракт `IBidHistoryRepository` экспортирует счетчик числа ставок для аукциона. Для добавления такой возможности создадим индекс, помогающий увеличить скорость выполнения запроса этих данных. Добавьте класс индекса `BidHistory_NumberOfBids`, представленный в листинге 21.76, в папку `Infrastructure`.

Листинг 21.76. Класс индекса `BidHistory_NumberOfBids`

```

using System;
using System.Collections.Generic;
using System.Linq;
using Raven.Client.Indexes;
using Raven.Client.Document;
using Raven.Abstractions.Indexing;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using DDDPPP.Chap21.RavenDBExample.Application.Application;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class BidHistory_NumberOfBids : AbstractIndexCreationTask<Bid,
                                           BidHistory_NumberOfBids.ReduceResult>
    {
        public class ReduceResult

```

```

{
    public Guid AuctionId { get; set; }
    public int Count { get; set; }
}

public BidHistory_NumberOfBids()
{
    Map = bids => from bid in bids
        select new {bid.AuctionId, Count = 1};

    Reduce = results => from result in results
        group result by result.AuctionId into g
        select new { AuctionId = g.Key,
            Count = g.Sum(x => x.Count) };
}
}
}

```

Реализация `BidHistoryRepository` — листинг 21.77 — напоминает реализацию для `NHibernate`, отличаясь только использованием индекса, созданного выше, для возврата числа ставок на аукционе. Обратите внимание, что в запросе вызывается оптимизация `WaitForNonStaleResultsAsOfNow`, так как `RavenDB` поддерживает шаблон потенциальной непротиворечивости и поток выполнения может блокироваться в моменты обновления индекса.

Листинг 21.77. Реализация репозитория хронологии ставок

```

using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
        private readonly IDocumentSession _documentSession;

        public BidHistoryRepository(IDocumentSession documentSession)
        {
            _documentSession = documentSession;
        }

        public int NoOfBidsFor(Guid auctionId)
        {
            var count = _documentSession.Query<BidHistory_NumberOfBids>().
                ReduceResult,
                    BidHistory_NumberOfBids>()
                .Customize(x => x.WaitForNonStaleResultsAsOfNow())
                .FirstOrDefault(x => x.AuctionId == auctionId)
                ?? new BidHistory_NumberOfBids().ReduceResult();
        }
    }
}

```

```

        return count.Count;
    }

    public void Add(Bid bid)
    {
        _documentSession.Store(bid);
    }

    public BidHistory FindBy(Guid auctionId)
    {
        var bids = _documentSession.Query<Bid>()
            .Customize(x =>
                x.WaitForNonStaleResultsAsOfNow())
            .Where(x => x.AuctionId == auctionId)
            .ToList();

        return new BidHistory(bids);
    }
}

```

Реализация `IAuctionRepository` проста и показана в листинге 21.78.

Листинг 21.78. Реализация репозитория аукциона

```

using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Application.Queries;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly IDocumentSession _documentSession;

        public AuctionRepository(IDocumentSession documentSession)
        {
            _documentSession = documentSession;
        }

        public void Add(Auction auction)
        {
            _documentSession.Store(auction);
        }

        public Auction FindBy(Guid Id)
        {
            return _documentSession.Load<Auction>("Auctions/" + Id);
        }
    }
}

```

Конфигурация

Чтобы обеспечить внедрение всех зависимостей в прикладные службы, задействуем StructureMap. Установите библиотеки StructureMap, как описывалось в примере для NHibernate, используя для этого NuGet. Затем добавьте следующий класс `Bootstrapper` (листинг 21.79) в корень проекта библиотеки классов.

Листинг 21.79. Класс `Bootstrapper` для настройки StructureMap

```
using System;
using System.Collections.Generic;
using System.Linq;
using Raven.Client;
using Raven.Client.Document;
using Raven.Client.Extensions;
using Raven.Client.Indexes;
using StructureMap;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.RavenDBExample.Application
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            var documentStore = new DocumentStore
            {
                ConnectionStringName = "RavenDB"
            }.Initialize();

            documentStore.DatabaseCommands
                .EnsureDatabaseExists("RepositoryExample");

            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                    .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();

                config.For<IDocumentStore>().Use(documentStore);
                config.For<IDocumentSession>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(x =>
                    {
                        var store = x.GetInstance<IDocumentStore>();
                        var session = store.OpenSession();
                        session.Advanced.UseOptimisticConcurrency = true;
                        return session;
                    });

                IndexCreation.CreateIndexes(typeof(BidHistory_NumberOfBids)
```

```

        .Assembly, documentStore);
    });
}
}
}

```

Наконец, необходимо определить некоторые настройки в файле `app.config`, в проекте `Presentation` консольного приложения, как показано в листинге 21.80.

Листинг 21.80. Конфигурационный файл `app.config` для RavenDB

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <connectionStrings>
    <add name="RavenDB"
      connectionString=
        "Url=http://localhost:8080;Database=RepositoryExample" />
  </connectionStrings>
</configuration>

```

Класс `Program` в проекте консольного приложения `Presentation` остался тем же, что в примере для `NHibernate`. После запуска примера программы можно открыть веб-приложение `RavenDB Management Studio` и исследовать получившийся документ, представляющий аукцион, как показано на рис. 21.17.

Фреймворк сохранения не способен отобразить предметную модель в модель данных без компромиссов

При использовании фреймворков, не способных отобразить предметную модель непосредственно в модель данных, следует использовать иной подход к организации хранилища. Все так же важно гарантировать отсутствие влияния модели данных на структуру предметной модели, чтобы последнюю можно было продолжать развивать независимо. Далее будут представлены два примера: один на основе фреймворка `Entity Framework`, другой — на основе низкоуровневого механизма `ADO.NET` с применением микрофреймворка объектно-реляционного отображения `Dapper`.

Пример для `Entity Framework`

`Entity Framework` — это инструмент ORM уровня предприятия от компании `Microsoft`. Он имеет схожие возможности с `NHibernate`, но справедливости ради следует отметить, что фреймворк `NHibernate` является более зрелым проектом, с более широкими возможностями непосредственного отображения предметных моделей. В этом примере фреймворк `Entity Framework` используется только для отображения модели данных, а для хранения моментального снимка предметной модели применяется шаблон «Хранитель» (`memento`).

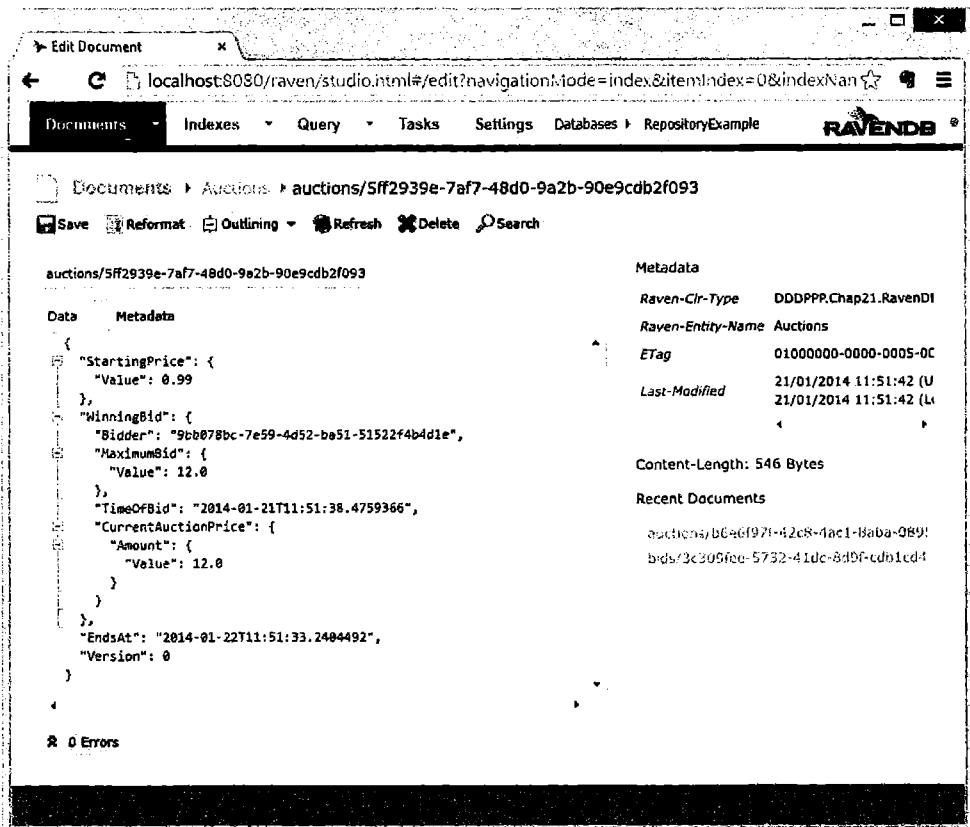


Рис. 21.17. Исследование содержимого документа в RavenDB Management Studio

Настройка решения

Создайте в Visual Studio новое пустое решение с именем DDDPPP.Chap21.EFExample и добавьте в него библиотеку классов DDDPPP.Chap21.EFExample.Application и консольное приложение DDDPPP.Chap21.EFExample.Presentation. Добавьте в проект Presentation ссылку на библиотеку Application. Файл Class1.cs, созданный автоматически, можно удалить, потому что нам он не понадобится. С помощью NuGet установите клиентские библиотеки Entity Framework, как показано на рис. 21.18.

Создайте приложение, как это описывается в примере для NHibernate, или просто скопируйте файлы классов в новое решение, не забыв при этом изменить названия пространств имен. Удалите следующие классы, специфические для примера на основе NHibernate:

- BusinessUseCases\BidOnAuction.cs
- BusinessUseCases\CreateAuction.cs
- Queries\AuctionStatusQuery.cs
- Queries\BidHistoryQuery.cs

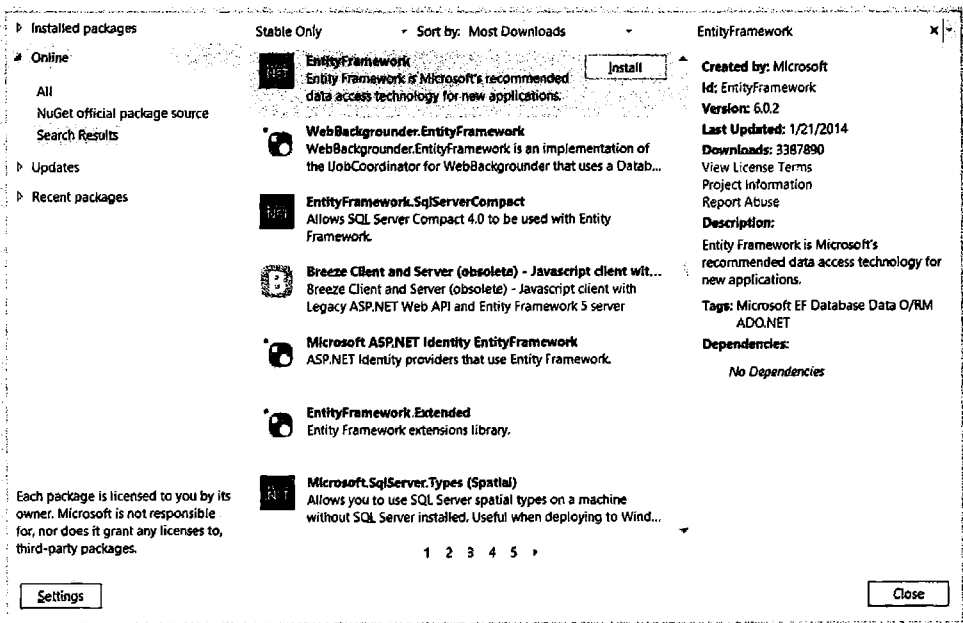


Рис. 21.18. Установка клиентских библиотек Entity Framework с помощью NuGet

- Infrastructure\Auction.hbm.xml
- Infrastructure\Bid.hbm.xml
- Infrastructure\AuctionRepository.cs
- Infrastructure\BidHistoryRepository.cs
- Bootstrapper.cs

После этого решение должно выглядеть, как показано на рис. 21.19.

Создайте также таблицы в базе данных, используя схему из примера для NHibernate, если вы этого еще не сделали.

Изменение модели

Так как мы собираемся реализовать шаблон «Хранитель» (memento), нам потребуется организовать получение моментальных снимков агрегатов, чтобы их можно было отобразить в модель данных. Сначала реализуем получение снимка для объекта-значения `WinningBid`. Используя листинг 21.81, добавьте в папку `Auction`, находящуюся в папке `Model`, новый класс `WinningBidSnapshot`.

Листинг 21.81. Класс `BidSnapshot`

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class WinningBidSnapshot
    {
        public Guid BiddersId { get; set; }
```

```
public DateTime TimeOfBid { get; set; }  
public decimal BiddersMaximumBid { get; set; }  
public decimal CurrentPrice { get; set; }  
}  
}
```

Создайте класс, представляющий моментальный снимок самого аукциона, как показано в листинге 21.82. Он будет хранить `WinningBidSnapshot` и представлять полный снимок агрегата `Auction`.

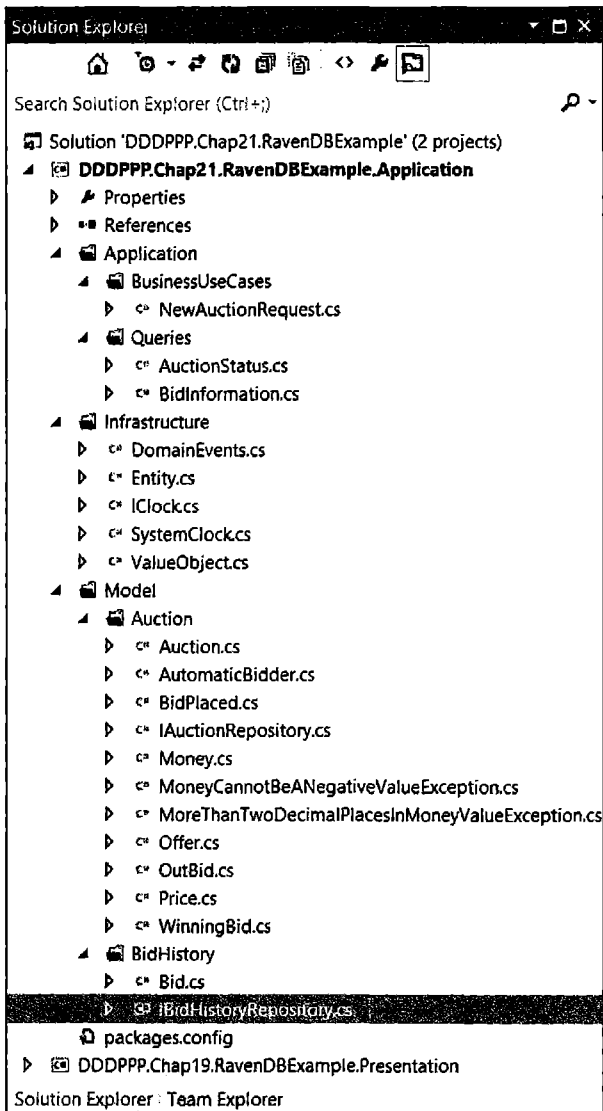


Рис. 21.19. Содержимое панели обозревателя решения с приложением, основанного на примере для NHibernate

Листинг 21.82. Класс AuctionSnapshot

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class AuctionSnapshot
    {
        public Guid Id { get; set; }
        public decimal StartingPrice { get; set; }
        public DateTime EndsAt { get; set; }
        public WinningBidSnapshot WinningBid { get; set; }
        public int Version { get; set; }
    }
}
```

Наконец, создайте класс, представляющий моментальный снимок объекта-значения Money, как показано в листинге 21.83.

Листинг 21.83. Класс MoneySnapshot

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class MoneySnapshot
    {
        public decimal Value { get; set; }
    }
}
```

Так как далее потребуется доступ к методам чтения и записи свойств Version и Id, переопределите соответствующие методы записи как защищенные, а методы чтения — как общедоступные, как показано в листинге 21.84.

Листинг 21.84. Базовый класс Entity

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }
        public int Version { get; protected set; }
    }
}
```

Для извлечения состояний предметных объектов в агрегате аукциона нужно добавить новые общедоступные методы, возвращающие моментальные снимки. Сначала добавим такой метод в объект-значение Money. Добавьте в класс Money новый метод GetSnapshot, как показано в листинге 21.85.

Листинг 21.85. Новый дополнительный метод в классе Money

```
namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class Money : ValueObject<Money>, IComparable<Money>
    {
        protected decimal Value { get; set; }

        public Money()
            : this(0m)
        {
        }

        // ....

        public MoneySnapshot GetSnapshot()
        {
            return new MoneySnapshot() { Value = this.Value };
        }
    }
}
```

Добавьте аналогичный метод в объект `WinningBid`, но так как это более сложный тип, добавьте также новый статический фабричный метод, позволяющий воссоздавать объект `WinningBid` из моментального снимка, как показано в листинге 21.86.

Листинг 21.86. Новые методы в классе `WinningBid`

```
namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
    {
        private WinningBid() { }

        public WinningBid(Guid bidder, Money maximumBid, Money bid,
            DateTime timeOfBid)
        {
            if (bidder == Guid.Empty)
                throw new ArgumentNullException("Bidder cannot be null");

            if (maximumBid == null)
                throw new ArgumentNullException("MaximumBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            Bidder = bidder;
            MaximumBid = maximumBid;
            TimeOfBid = timeOfBid;
            CurrentAuctionPrice = new Price(bid);
        }

        // ....

        public WinningBidSnapshot GetSnapshot()
        {
        }
    }
}
```

```

        var snapshot = new WinningBidSnapshot();

        snapshot.BiddersId = this.Bidder;
        snapshot.BiddersMaximumBid = this.MaximumBid.GetSnapshot().Value;
        snapshot.CurrentPrice =
            this.CurrentAuctionPrice.Amount.GetSnapshot().Value;
        snapshot.TimeOfBid = this.TimeOfBid;

        return snapshot;
    }

    public static WinningBid CreateFrom(WinningBidSnapshot bidSnapshot)
    {
        return new WinningBid(bidSnapshot.BiddersId,
                               new Money(bidSnapshot.BiddersMaximumBid),
                               new Money(bidSnapshot.CurrentPrice),
                               bidSnapshot.TimeOfBid);
    }
}

```

Наконец, добавьте в класс `Auction` статический фабричный метод, извлекающий экземпляр аукциона из моментального снимка, и метод получения снимка, как показано в листинге 21.87.

Листинг 21.87. Новые методы в классе `Auction`

```

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        public Auction(Guid id, Money startingPrice, DateTime endsAt)
        {
            if (id == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (startingPrice == null)
                throw new ArgumentNullException("Starting Price cannot be null");

            if (endsAt == DateTime.MinValue)
                throw new ArgumentNullException("EndsAt must have a value");

            Id = id;
            StartingPrice = startingPrice;
            EndsAt = endsAt;
        }

        private Auction(AuctionSnapshot snapshot)
        {
            this.Id = snapshot.Id;
            this.StartingPrice = new Money(snapshot.StartingPrice);
            this.EndsAt = snapshot.EndsAt;
            this.Version = snapshot.Version;
        }
    }
}

```

```

        if (snapshot.WinningBid != null)
            WinningBid = WinningBid.CreateFrom(snapshot.WinningBid);
    }

    public static Auction CreateFrom(AuctionSnapshot snapshot)
    {
        return new Auction(snapshot);
    }

    private Money StartingPrice { get; set; }
    private WinningBid CurrentWinningBid { get; set; }
    private DateTime EndsAt { get; set; }

    public AuctionSnapshot GetSnapshot()
    {
        var snapshot = new AuctionSnapshot();

        snapshot.Id = this.Id;
        snapshot.StartingPrice = this.StartingPrice.GetSnapshot().Value;
        snapshot.EndsAt = this.EndsAt;
        snapshot.Version = this.Version;

        if (HasACurrentBid())
            snapshot.WinningBid = WinningBid.GetSnapshot();

        return snapshot;
    }

    private bool HasACurrentBid()
    {
        return WinningBid != null;
    }

    // ....

```

Требуется также добавить объект `BidHistory`, как в примере для RavenDB. Определение объекта приводится в листинге 21.88.

Листинг 21.88. Класс `BidHistory`

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.EFExample.Application.Model.BidHistory
{
    public class BidHistory
    {
        private IEnumerable<Bid> _bids;

        public BidHistory(IEnumerable<Bid> bids)

```

```

    {
        if (bids == null)
            throw new ArgumentNullException("Bids cannot be null");

        _bids = bids;
    }

    public IEnumerable<Bid> ShowAllBids()
    {
        var bids = _bids.OrderByDescending(x => x.AmountBid)
                           .ThenBy(x => x.TimeOfBid);

        return bids;
    }
}

```

Чтобы получить возможность извлекать объекты `BidHistory`, нужно дополнить определение интерфейса `IBidHistoryRepository`, как показано в листинге 21.89.

Листинг 21.89. Интерфейс репозитория хронологии ставок

```

using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.BidHistory
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid auctionId);
        void Add(Bid bid);
        BidHistory FindBy(Guid auctionId);
    }
}

```

Поскольку в данном примере отсутствует поддержка автоматического слежения за изменениями в предметных объектах, нужно явно сохранять их, то есть необходимо добавить метод сохранения в интерфейс `IAuctionRepository`, как показано в листинге 21.90.

Листинг 21.90. Интерфейс репозитория аукциона

```

using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public interface IAuctionRepository
    {
        void Add(Auction auction);
        void Save(Auction auction);
        Auction FindBy(Guid Id);
    }
}

```


Реализация репозитория Entity Framework

Создайте новую папку `DataModel` в папке `Infrastructure`. Здесь будут храниться определения объектов из модели данных, отображаемых в таблицы и записи базы данных. Первый объект, который мы создадим, представляет запись в таблице `Auctions` и определяется в листинге 21.91.

Листинг 21.91. Объект переноса данных об аукционе

```
using System;
using System.Collections.Generic;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel
{
    public partial class AuctionDTO
    {
        public System.Guid Id { get; set; }
        public decimal StartingPrice { get; set; }
        public DateTime AuctionEnds { get; set; }
        public Nullable<System.Guid> BidderMemberId { get; set; }
        public System.DateTime? TimeOfBid { get; set; }
        public Nullable<decimal> MaximumBid { get; set; }
        public Nullable<decimal> CurrentPrice { get; set; }
        public int Version { get; set; }
    }
}
```

Второй аналогичный объект (листинг 21.92) представляет запись в таблице `BidHistory`.

Листинг 21.92. Объект переноса данных о ставке

```
using System;
using System.Collections.Generic;
namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel
{
    public partial class BidDTO
    {
        public System.Guid Id { get; set; }
        public System.Guid AuctionId { get; set; }
        public System.Guid BidderId { get; set; }
        public decimal Bid { get; set; }
        public System.DateTime TimeOfBid { get; set; }
    }
}
```

Для отображения модели данных в базу данных мы воспользуемся такой особенностью Entity Framework, как преимущество соглашений перед настройками (code-first). Добавьте новую папку `Mapping` в папку `Infrastructure`. Класс `AuctionMap` (в листинге 21.93) отображает `AuctionDTO` в таблицу `Auctions`.

Листинг 21.93. Класс AuctionMap

```

using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping
{
    public class AuctionMap : EntityTypeConfiguration<AuctionDTO>
    {
        public AuctionMap()
        {
            this.HasKey(t => t.Id);

            this.ToTable("Auctions");
            this.Property(t => t.Id).HasColumnName("Id");
            this.Property(t => t.StartingPrice).HasColumnName("StartingPrice");
            this.Property(t =>
                t.BidderMemberId).HasColumnName("BidderMemberId");
            this.Property(t => t.TimeOfBid).HasColumnName("TimeOfBid");
            this.Property(t => t.MaximumBid).HasColumnName("MaximumBid");
            this.Property(t => t.CurrentPrice).HasColumnName("CurrentPrice");
            this.Property(t => t.AuctionEnds).HasColumnName("AuctionEnds");
            this.Property(t =>
                t.Version).HasColumnName("Version").IsConcurrencyToken();
        }
    }
}

```

Аналогично, класс BidMap (листинг 21.94) отображает BidDTO в таблицу BidHistory.

Листинг 21.94. Класс BidMap

```

using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping
{
    public class BidMap : EntityTypeConfiguration<BidDTO>
    {
        public BidMap()
        {
            this.HasKey(t => t.Id);

            this.ToTable("BidHistory");
            this.Property(t => t.Id).HasColumnName("Id");
            this.Property(t => t.AuctionId).HasColumnName("AuctionId");
            this.Property(t => t.BidderId).HasColumnName("BidderId");
            this.Property(t => t.Bid).HasColumnName("Bid");
            this.Property(t => t.TimeOfBid).HasColumnName("TimeOfBid");
        }
    }
}

```

Для взаимодействий с базой данных требуется контекст, представленный в листинге 21.95. Объект `DbContext` в Entity Framework напоминает `ISession` в NHibernate и `IDocumentSession` в RavenDB. Фактически это реализация шаблона «Единая работа» в Entity Framework.

Листинг 21.95. Класс `AuctionDatabaseContext` для взаимодействий с Entity Framework

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public partial class AuctionDatabaseContext : DbContext
    {
        static AuctionDatabaseContext()
        {
            Database.SetInitializer<AuctionDatabaseContext>(null);
        }

        public AuctionDatabaseContext()
            : base("Name=AuctionDatabaseContext")
        {
        }

        public DbSet<AuctionDTO> Auctions { get; set; }
        public DbSet<BidDTO> Bids { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new AuctionMap());
            modelBuilder.Configurations.Add(new BidMap());
        }

        public void Clear()
        {
            var context = ((IObjContextAdapter)this).ObjectContext;

            var addedObjects = context
                .ObjectStateManager
                .GetObjectStateEntries(EntityState.Added);

            foreach (var objectStateEntry in addedObjects)
            {
                context.Detach(objectStateEntry.Entity);
            }

            var modifiedObjects = context
                .ObjectStateManager
                .GetObjectStateEntries(EntityState.Modified);
```

```

        foreach (var objectStateEntry in modifiedObjects)
        {
            context.Detach(objectStateEntry.Entity);
        }
    }
}
}

```

Реализация репозитория в этом примере отличается от реализаций, показанных в примерах для NHibernate и RavenDB. Из-за того что отсутствует возможность отобразить предметную модель непосредственно в модель данных, репозитории должны извлекать и отображать сами моментальные снимки. Реализация `AuctionRepository` показана в листинге 21.96.

Листинг 21.96. Реализация репозитория аукциона

```

using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Application;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly AuctionDatabaseContext _auctionExampleContext;

        public AuctionRepository(AuctionDatabaseContext auctionExampleContext)
        {
            _auctionExampleContext = auctionExampleContext;
        }

        public void Add(Auction auction)
        {
            var auctionDTO = new AuctionDTO();
            Map(auctionDTO, auction.GetSnapshot());
            _auctionExampleContext.Auctions.Add(auctionDTO);
        }

        public void Save(Auction auction)
        {
            var auctionDTO = _auctionExampleContext.Auctions.Find(auction.Id);
            Map(auctionDTO, auction.GetSnapshot());
        }

        public Auction FindBy(Guid Id)
        {
            var auctionDTO = _auctionExampleContext.Auctions.Find(Id);
            var auctionSnapshot = new AuctionSnapshot();

```

Работа репозитория `BidHistoryRepository` определяется строками в листинге 21.97.

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;
```

```
namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
```

```

private readonly AuctionDatabaseContext _auctionExampleContext;

public BidHistoryRepository(
    AuctionDatabaseContext auctionExampleContext)
{
    _auctionExampleContext = auctionExampleContext;
}

public int NoOfBidsFor(Guid auctionId)
{
    return _auctionExampleContext.Bids
        .Count(x => x.AuctionId == auctionId);
}

public void Add(Bid bid)
{
    var bidDTO = new BidDTO();

    bidDTO.AuctionId = bid.AuctionId;
    bidDTO.Bid = bid.AmountBid.GetSnapshot().Value;
    bidDTO.BidderId = bid.Bidder;
    bidDTO.TimeOfBid = bid.TimeOfBid;

    bidDTO.Id = Guid.NewGuid();

    _auctionExampleContext.Bids.Add(bidDTO);
}

public BidHistory FindBy(Guid auctionId)
{
    var bidDTOS = _auctionExampleContext.Bids.Where<BidDTO>(x =>
        x.AuctionId == auctionId).ToList();
    var bids = new List<Bid>();

    foreach (var bidDTO in bidDTOS)
    {
        bids.Add(new Bid(bidDTO.AuctionId, bidDTO.BidderId,
            new Money(bidDTO.Bid), bidDTO.TimeOfBid));
    }

    return new BidHistory(bids);
}
}

```

Прикладные службы

Прикладные службы в этом примере мало чем отличаются от тех, что были показаны выше, за исключением использования `DbContext` — реализации шаблона «Единая работа» в Entity Framework. Единственное, что следует отметить, — Entity Framework неявно заворачивает вызов `SaveChanges` в транзакцию, поэтому нам делать этого не нужно. Необходимые изменения представлены в листинге 21.98.

Листинг 21.98. Прикладная служба создания аукциона

```

using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;

namespace DDDPPP.Chap21.EFExample.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctions;
        private AuctionDatabaseContext _unitOfWork;
        public CreateAuction(IAuctionRepository auctions,

        AuctionDatabaseContext unitOfWork)
        {
            _auctions = auctions;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            _auctions.Add(new Auction(auctionId, startingPrice,
                                     command.EndsAt));

            _unitOfWork.SaveChanges();

            return auctionId;
        }
    }
}

```

В прикладной службе `BidOnAuction`, представленной в листинге 21.99, нужно явно вызвать метод `Save` репозитория. Еще одно отличие заключается в типе исключения, которое возбуждается, когда возникает конфликт при одновременном доступе, — он отличается от типов, использовавшихся в примерах для NHibernate и RavenDB.

Листинг 21.99. Прикладная служба предложения ставки на аукционе

```

using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using System.Data.Entity.Infrastructure;

```

```

namespace DDDPPP.Chap21.EFExample.Application.Application.BusinessUseCases
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctions;
        private IBidHistoryRepository _bidHistory;
        private AuctionDatabaseContext _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctions,
                            IBidHistoryRepository bidHistory,
                            AuctionDatabaseContext unitOfWork, IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctions.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                                  _clock.Time(), _clock.Time()));
                    _auctions.Save(auction);
                }

                _unitOfWork.SaveChanges();
            }
            catch (DbUpdateConcurrencyException ex)
            {
                _unitOfWork.Clear();

                Bid(auctionId, memberId, amount);
            }
        }

        private Action<BidPlaced> BidPlaced()
        {
            return (BidPlaced e) =>
            {
                var bidEvent = new Bid(e.AuctionId, e.Bidder,
                                       e.AmountBid, e.TimeOfBid);
                _bidHistory.Add(bidEvent);
            };
        }
    }
}

```



```

    }

    private Action<OutBid> OutBid()
    {
        return (OutBid e) =>
        {
            // Послать электронное письмо участнику с сообщением,
            // что его ставка перебита
        };
    }
}
}
}

```

Справочные службы

Для заполнения моделей представлений в реализациях справочных служб `AuctionStatusQuery` и `BidHistoryQuery` можно использовать моментальные снимки, как показано в листингах 21.100 и 21.101 соответственно.

Листинг 21.100. Справочная служба `AuctionStatusQuery`

```

using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;

namespace DDDPPP.Chap21.EFExample.Application.Application.Queries
{
    public class AuctionStatusQuery
    {
        private readonly IAuctionRepository _auctions;
        private readonly IBidHistoryRepository _bidHistory;
        private readonly IClock _clock;

        public AuctionStatusQuery(IAuctionRepository auctions,
                                   IBidHistoryRepository bidHistory,
                                   IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _clock = clock;
        }

        public AuctionStatus AuctionStatus(Guid auctionId)
        {
            var auction = _auctions.FindBy(auctionId);

            var snapshot = auction.GetSnapshot();

            return ConvertToStatus(snapshot);
        }

        public AuctionStatus ConvertToStatus(AuctionSnapshot snapshot)

```

```

{
    var status = new AuctionStatus();

    status.AuctionEnds = snapshot.EndsAt;
    status.Id = snapshot.Id;
    status.TimeRemaining = TimeRemaining(snapshot.EndsAt);

    if (snapShot.WinningBid != null)
    {
        status.NumberOfBids = _bidHistory.NoOfBidsFor(snapshot.Id);
        status.WinningBidderId = snapshot.WinningBid.BiddersId;
        status.CurrentPrice = snapshot.WinningBid.CurrentPrice;
    }

    return status;
}

public TimeSpan TimeRemaining(DateTime AuctionEnds)
{
    if (_clock.Time() < AuctionEnds)
        return AuctionEnds.Subtract(_clock.Time());
    else
        return new TimeSpan();
}
}
}

```

Листинг 21.101. Справочная служба BidHistroyQuery

```

using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.EFExample.Application.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly IBidHistory _bidHistory;

        public BidHistoryQuery(IBidHistory bidHistory)
        {
            _bidHistory = bidHistory;
        }

        public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
        {
            var bidHistory = _bidHistory.FindBy(auctionId);

            return Convert(bidHistory.ShowAllBids());
        }
    }
}

```

```

    }

    public IEnumerable<BidInformation> Convert(IEnumerable<BidEvent> bids)
    {
        var bidInfo = new List<BidInformation>();

        foreach (var bid in bids)
        {
            bidInfo.Add(new BidInformation() { Bidder = bid.Bidder,
                                                AmountBid = bid.AmountBid.GetSnapshot().Value,
                                                TimeOfBid = bid.TimeOfBid });
        }

        return bidInfo;
    }
}

```

Конфигурация

Для внедрения зависимостей в этом примере снова будет использоваться контейнер StructureMap, поэтому его нужно установить с помощью NuGet и создать класс инициализации в соответствии с листингом 21.102.

Листинг 21.102. Класс Bootstrapper для инициализации StructureMap

```

using System;
using StructureMap;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.EFExample.Application
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                    .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();
            });
        }
    }
}

```

Наконец, добавьте фрагмент XML из листинга 21.103 в файл `app.config`, находящийся в проекте `Presentation` консольного приложения.

Листинг 21.103. Настройки для Entity Framework

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- За дополнительной информацией о настройках Entity Framework
         обращайтесь по адресу http://go.microsoft.com/fwlink/?LinkID=237468
    -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
        EntityFramework, Version=6.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
      requirePermission="false" />

    <!-- За дополнительной информацией о настройках Entity Framework
         обращайтесь по адресу http://go.microsoft.com/fwlink/?LinkID=237468
    -->
  </configSections>
  <connectionStrings>
    <add name="AuctionDatabaseContext"
      connectionString="DataSource=.\sqlexpress;Initial
        Catalog=AuctionExample;
        IntegratedSecurity=True;
        MultipleActiveResultSets=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
  <entityFramework>
    <defaultConnectionFactory
      type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
        EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
        type="System.Data.Entity.SqlServer.SqlProviderServices,
          EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
</configuration>

```

Если теперь запустить консольное приложение, вы получите те же результаты, что и в примерах для NHibernate и RavenDB.

Пример для Micro ORM

Даргер — это микрофреймворк объектно-реляционного отображения, он помогает отобразить записи из базы данных в объекты — и ничего другого. В этом примере нам потребуется вручную реализовать шаблон «Единица работы» и проверку конфликтов при одновременном доступе.

Настройка решения

Для начала создайте в Visual Studio новое пустое решение с именем DDDPPP.Chap21. MicroORM, добавьте в него библиотеку классов DDDPPP.Chap21. MicroORM. Application и консольное приложение DDDPPP.Chap21. MicroORM. Presentation. В проекте Presentation добавьте ссылку на библиотеку Application. Файл Class1.cs, созданный автоматически, можно удалить, потому что нам он не понадобится. С помощью NuGet установите клиентские библиотеки Dapper, как показано на рис. 21.20.

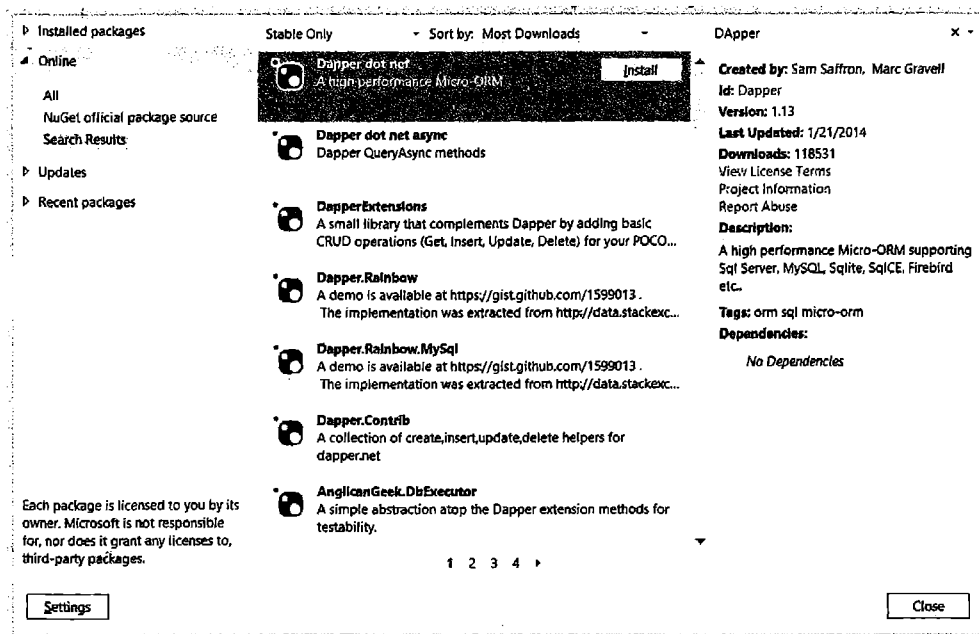


Рис. 21.20. Установка клиентских библиотек Dapper с помощью NuGet

Предметная модель останется прежней, как в примере для Entity Framework. Прежней также останется схема базы данных, как в примерах для NHibernate и Entity Framework. Создайте приложение, как это описывается в примере для Entity Framework, но исключите следующие файлы:

- BusinessUseCases/BidOnAuction.cs
- BusinessUseCases/CreateAuction.cs
- Infrastructure/Mapping
- Infrastructure/AuctionDatabaseContext.cs
- Infrastructure/AuctionRepository.cs
- Infrastructure/BidHistoryRepository.cs
- Bootstrapper.cs

После этого решение должно выглядеть, как показано на рис. 21.21.

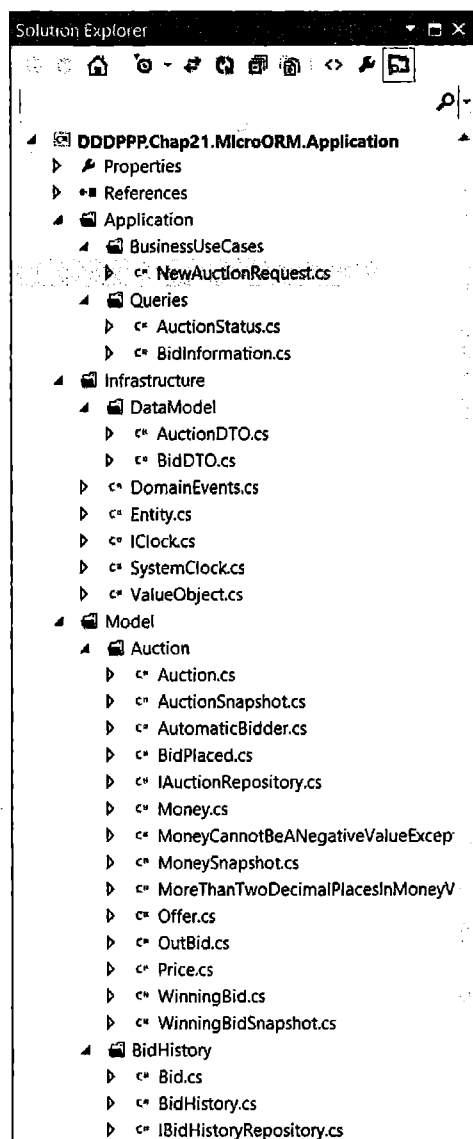


Рис. 21.21. Структура решения на основе Entity Framework в Visual Studio

Инфраструктура

В этом примере используется микрофреймворк объектно-реляционного отображения (micro ORM). Главной целью таких микрофреймворков обычно является простота и скорость, так как они меньше перегружены разнообразными особенностями в отличие от их полностью автоматизированных собратьев Entity Framework и NHibernate. Как следствие, нам придется вручную реализовать ша-

блон «Единица работы». Структура единицы работы в этом примере основана на структуре, предложенной Тимом Маккарти (Tim McCarthy) в его книге «.NET Domain-Driven Design with C#: Problem-Design-Solution».

Интерфейс `IAggregateDataModel` (листинг 21.104) фактически сам является шаблоном, получившим название «Интерфейс-маркер» (marker interface). Интерфейс действует как метаданные для класса, и методы, взаимодействующие с экземплярами такого класса, проверяют поддержку интерфейса, прежде чем выполнить свою работу. Вы увидите этот шаблон в работе далее, когда мы будем создавать уровень репозитория, который сохраняет только предметные объекты, реализующие интерфейс `IAggregateDataModel`.

Листинг 21.104. Интерфейс `IAggregateDataModel`

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IAggregateDataModel
    {
    }
}
```

Реализация единицы работы использует интерфейс `IAggregateDataModel` для ссылки на любые предметные сущности, принимающие участие в транзакции. Добавьте в проект `Infrastructure` еще один интерфейс с именем `IUnitOfWorkRepository` (листинг 21.105), определяющий контракт.

Листинг 21.105. Интерфейс `IUnitOfWorkRepository`

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IUnitOfWorkRepository
    {
        void PersistCreationOf(IAggregateDataModel entity);
        void PersistUpdateOf(IAggregateDataModel entity);
    }
}
```

`IUnitOfWorkRepository` — это второй интерфейс, обязательный для реализации во всех репозиториях, которые предполагается использовать в единице работы. Определение этого контракта можно было бы добавить в модель интерфейса `Repository`, который будет определен позже, но дело в том, что эти интерфейсы решают разные задачи. Это соответствует принципу разделения интерфейсов (Interface Segregation Principle). Мы не собираемся ничего удалять в нашем приложении, поэтому нет необходимости добавлять данный метод.

Наконец, добавьте в проект `Infrastructure` третий интерфейс с именем `IUnitOfWork`, определение которого приводится в листинге 21.106.

Листинг 21.106. Интерфейс `IUnitOfWork`

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IUnitOfWork
    {
        void RegisterAmended(IAggregateDataModel entity,
                             IUnitOfWorkRepository unitOfWorkRepository);
        void RegisterNew(IAggregateDataModel entity,
                         IUnitOfWorkRepository unitOfWorkRepository);
        void Commit();
        void Clear();
    }
}
```

Интерфейс `IUnitOfWork` требует наличия поддержки интерфейса `IUnitOfWorkRepository` при регистрации операций изменения/добавления/удаления, чтобы во время подтверждения транзакции единица работы могла делегировать работу фактическому методу работы с хранилищем в соответствующей реализации. Логика методов интерфейса `IUnitOfWork` станет более понятна, когда вы увидите реализацию интерфейса `IUnitOfWork` по умолчанию и приступите к созданию репозитория.

Наконец, нужно создать класс исключения `ConcurrencyException` (листинг 21.107), которое будет возбуждаться, если аукцион изменился между моментами его извлечения, добавления ставки и сохранения.

Листинг 21.107. Класс `ConcurrencyException`

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class ConcurrencyException : ApplicationException
    {
    }
}
```

Прикладные службы

К настоящему моменту обе прикладные службы должны быть уже хорошо знакомы вам. В листингах 21.108 и 21.109 приводятся изменения, обусловленные отличиями в реализации единицы работы и типе исключения, возбуждаемого в случае конфликтов при одновременном доступе.

Листинг 21.108. Прикладная служба создания аукциона

```
using System;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;
```



```
namespace DDDPPP.Chap21.MicroORM.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctionRepository;
        private IUnitOfWork _unitOfWork;

        public CreateAuction(IAuctionRepository auctionRepository,
                             IUnitOfWork unitOfWork)
        {
            _auctionRepository = auctionRepository;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            _auctionRepository.Add(new Auction(auctionId,
                                                startingPrice, command.EndsAt));

            _unitOfWork.Commit();

            return auctionId;
        }
    }
}
```

Листинг 21.109. Прикладная служба размещения ставки в аукционе

```
using System;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;

namespace DDDPPP.Chap21.MicroORM.Application.Application.BusinessUseCases
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctionRepository;
        private IBidHistoryRepository _bidHistoryRepository;
        private IUnitOfWork _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctionRepository,
                             IBidHistoryRepository bidHistoryRepository,
                             IUnitOfWork unitOfWork, IClock clock)
        {
            _auctionRepository = auctionRepository;
            _bidHistoryRepository = bidHistoryRepository;
            _unitOfWork = unitOfWork;
        }
    }
}
```

```

        _clock = clock;
    }

    public void Bid(Guid auctionId, Guid memberId, decimal amount)
    {
        try
        {
            using (DomainEvents.Register(OutBid()))
            using (DomainEvents.Register(BidPlaced()))
            {
                var auction = _auctionRepository.FindBy(auctionId);

                var bidAmount = new Money(amount);

                auction.PlaceBidFor(new Offer(memberId, bidAmount,

                    _clock.Time(), _clock.Time()));
                _auctionRepository.Save(auction);
            }

            _unitOfWork.Commit();
        }
        catch (ConcurrencyException ex)
        {
            _unitOfWork.Clear();

            Bid(auctionId, memberId, amount);
        }
    }

    private Action<BidPlaced> BidPlaced()
    {
        return (BidPlaced e) =>
        {
            var bidEvent = new Bid(e.AuctionId, e.Bidder, e.AmountBid,
                e.TimeOfBid);

            _bidHistoryRepository.Add(bidEvent);
        };
    }

    private Action<OutBid> OutBid()
    {
        return (OutBid e) =>
        {
            // Послать электронное письмо участнику с сообщением,
            // что его ставка перебита
        };
    }
}
}
}

```

Реализация репозитория на основе ADO.NET

Нам нужно добавить в объекты AuctionDTO (листинг 21.110) и BidDTO (листинг 21.111) реализацию интерфейса IAggregateDataModel, так как экземпляры этих классов будут сохраняться с использованием реализации единицы работы.

Листинг 21.110. Дополнения в AuctionDTO для реализации IAggregateDataModel

```
namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel
{
    public partial class AuctionDTO : IAggregateDataModel
    {
        // .....
    }
}
```

Листинг 21.111. Дополнения в BidDTO для реализации IAggregateDataModel

```
namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel
{
    public partial class BidDTO : IAggregateDataModel
    {
        // .....
    }
}
```

Оба репозитория, AuctionRepository и BidHistoryRepository, показанные в листингах 21.112 и 21.113, реализуют контракты репозитория предметной модели IAuctionRepository и IBidHistoryRepository, а также интерфейс IUnitOfWorkRepository. Реализации обоих репозиториях просто делегируют выполнение операций единице работы, передавая сущность для сохранения вместе со ссылкой на репозиторий, который, конечно же, реализует IUnitOfWorkRepository. Как мы увидим далее, при вызове метода Commit единица работы ссылается на реализацию контракта IUnitOfWorkRepository для выполнения фактических операций с хранилищем. Похожее поведение мы наблюдали в крупных фреймворках. Так как репозитории получают строку соединения из файла app.config, необходимо добавить ссылку на сборку Systems.Configuration, как показано на рис. 21.22.

Листинг 21.112. Реализация репозитория аукциона

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Application;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel;
using System.Data.SqlClient;
using Dapper;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
```

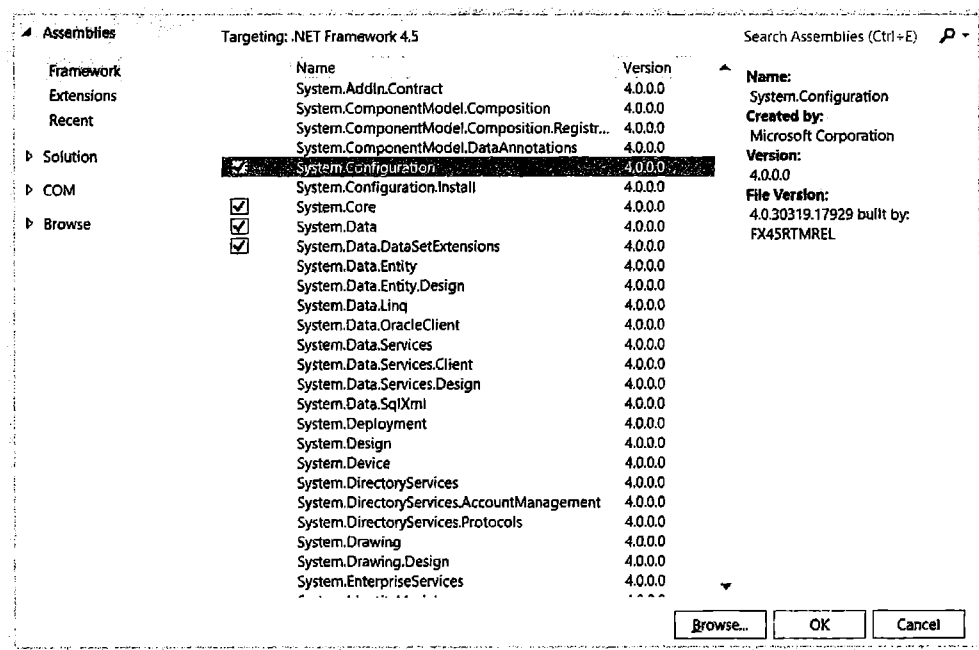


Рис. 21.22. Добавление ссылки на сборку Systems.Configuration

```
public class AuctionRepository : IAuctionRepository, IUnitOfWorkRepository
{
    private IUnitOfWork _unitOfWork;

    public AuctionRepository(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    public void Add(Auction auction)
    {
        var snapshot = auction.GetSnapshot();
        var auctionDTO = new AuctionDTO();

        Map(auctionDTO, snapshot);

        _unitOfWork.RegisterNew(auctionDTO, this);
    }

    public void Save(Auction auction)
    {
        var snapshot = auction.GetSnapshot();
        var auctionDTO = new AuctionDTO();

        Map(auctionDTO, snapshot);
```

```
        _unitOfWork.RegisterAmended(auctionDTO, this);
    }

    public Auction FindBy(Guid Id)
    {
        AuctionDTO auctionDTO;

        using (var connection = new
            SqlConnection(System.Configuration.ConfigurationManager
                .ConnectionStrings["AuctionDB"].ConnectionString))
        {
            auctionDTO = connection.Query<AuctionDTO>("Select * From " +
                "Auctions Where Id = CAST(@Id AS uniqueidentifier)"
                , new { Id = Id }).FirstOrDefault();
        }

        var auctionSnapshot = new AuctionSnapshot();

        auctionSnapshot.Id = auctionDTO.Id;
        auctionSnapshot.EndsAt = auctionDTO.AuctionEnds;
        auctionSnapshot.StartingPrice = auctionDTO.StartingPrice;
        auctionSnapshot.Version = auctionDTO.Version;

        if (auctionDTO.BidderMemberId.HasValue)
        {
            var bidSnapshot = new WinningBidSnapshot();

            bidSnapshot.BiddersMaximumBid = auctionDTO.MaximumBid.Value;
            bidSnapshot.CurrentPrice = auctionDTO.CurrentPrice.Value;
            bidSnapshot.BiddersId = auctionDTO.BidderMemberId.Value;
            bidSnapshot.TimeOfBid = auctionDTO.TimeOfBid.Value;
            auctionSnapshot.WinningBid = bidSnapshot;
        }

        return Auction.CreateFrom(auctionSnapshot);
    }

    public void PersistCreationOf(IAggregateDataModel entity)
    {
        var auctionDTO = (AuctionDTO)entity;

        using (var connection = new
            SqlConnection(System.Configuration.ConfigurationManager
                .ConnectionStrings["AuctionDB"].ConnectionString))
        {
            var recordsAdded = connection.Execute(@"
                INSERT INTO [AuctionExample].[dbo].[Auctions]
                ([Id]
                , [StartingPrice]
                , [BidderMemberId]
                , [TimeOfBid]
```

```

        , [MaximumBid]
        , [CurrentPrice]
        , [AuctionEnds]
        , [Version])
VALUES
    (@Id, @StartingPrice, @BidderMemberId, @TimeOfBid,
     @MaximumBid, @CurrentPrice, @AuctionEnds, @Version)"
, new { Id = auctionDTO.Id, StartingPrice =
        auctionDTO.StartingPrice,
        BidderMemberId = auctionDTO.BidderMemberId,
        TimeOfBid = auctionDTO.TimeOfBid,
        MaximumBid = auctionDTO.MaximumBid,
        CurrentPrice = auctionDTO.CurrentPrice,
        AuctionEnds = auctionDTO.AuctionEnds,
        Version = auctionDTO.Version });
    }
}

public void PersistUpdateOf(IAggregateDataModel entity)
{
    var auctionDTO = (AuctionDTO)entity;

    using (var connection = new
        SqlConnection(System.Configuration.ConfigurationManager
            .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        var recordsUpdated = connection.Execute(@"
UPDATE
    [AuctionExample].[dbo].[Auctions]
SET
    [Id] = @Id
    , [StartingPrice] = @StartingPrice
    , [BidderMemberId] = @BidderMemberId
    , [TimeOfBid] = @TimeOfBid
    , [MaximumBid] = @MaximumBid
    , [CurrentPrice] = @CurrentPrice
    , [AuctionEnds] = @AuctionEnds
    , [Version] = @Version
WHERE
    Id = @Id AND Version = @PreviousVersion"
, new
{
    Id = auctionDTO.Id,
    StartingPrice = auctionDTO.StartingPrice,
    BidderMemberId = auctionDTO.BidderMemberId,
    TimeOfBid = auctionDTO.TimeOfBid,
    MaximumBid = auctionDTO.MaximumBid,
    CurrentPrice = auctionDTO.CurrentPrice,
    AuctionEnds = auctionDTO.AuctionEnds,
    Version = auctionDTO.Version + 1,
    PreviousVersion = auctionDTO.Version

```

```

    });

    if (!recordsUpdated.Equals(1))
    {
        throw new ConcurrencyException();
    }
}

}

public void Map(AuctionDTO auctionDTO, AuctionSnapshot snapshot)
{
    auctionDTO.Id = snapshot.Id;
    auctionDTO.StartingPrice = snapshot.StartingPrice;
    auctionDTO.AuctionEnds = snapshot.EndsAt;
    auctionDTO.Version = snapshot.Version;

    if (snapshot.WinningBid != null)
    {
        auctionDTO.BidderMemberId = snapshot.WinningBid.BiddersId;
        auctionDTO.CurrentPrice = snapshot.WinningBid.CurrentPrice;
        auctionDTO.MaximumBid = snapshot.WinningBid.BiddersMaximumBid;
        auctionDTO.TimeOfBid = snapshot.WinningBid.TimeOfBid;
    }
}
}
}
}
}

```

Листинг 21.113. Реализация репозитория хронологии ставок

```

using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel;
using Dapper;
using System.Data.SqlClient;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository,
                                      IUnitOfWorkRepository
    {
        private IUnitOfWork _unitOfWork;

        public BidHistoryRepository(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public int NoOfBidsFor(Guid auctionId)
        {

```

```

    int count;

    using (var connection = new
        SqlConnection(System.Configuration.ConfigurationManager
            .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        var count1 = connection.Query<int>(
            "Select Count(*) From BidHistory Where AuctionId = @Id",
            new { Id = auctionId }).FirstOrDefault();

        count = count1 != null ? count1 : 1;
    }

    return count;
}

public void Add(Bid bid)
{
    var bidHistoryDTO = new BidDTO();

    bidHistoryDTO.AuctionId = bid.AuctionId;
    bidHistoryDTO.Bid = bid.AmountBid.GetSnapshot().Value;
    bidHistoryDTO.BidderId = bid.Bidder;
    bidHistoryDTO.TimeOfBid = bid.TimeOfBid;

    bidHistoryDTO.Id = Guid.NewGuid();

    _unitOfWork.RegisterNew(bidHistoryDTO, this);
}

public BidHistory FindBy(Guid auctionId)
{
    IEnumerable<BidDTO> bidDTOs;

    using (var connection = new
        SqlConnection(System.Configuration.ConfigurationManager
            .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        bidDTOs = connection.Query<BidDTO>(
            "Select * From BidHistory Where AuctionId = @Id",
            new { Id = auctionId });
    }

    var bids = new List<Bid>();

    foreach (var bid in bidDTOs)
    {
        bids.Add(new Bid(bid.AuctionId, bid.BidderId,
            new Money(bid.Bid), bid.TimeOfBid));
    }

    return new BidHistory(bids);
}

public void PersistCreationOf(IAggregateDataModel entity)

```



```

{
    var bidHistoryDTO = (BidDTO)entity;

    using (var connection = new
        SqlConnection(System.Configuration.ConfigurationManager
            .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        connection.Execute(@"
            INSERT INTO [dbo].[BidHistory]
                ([AuctionId]
                ,[BidderId]
                ,[Bid]
                ,[TimeOfBid]
                ,[Id])
            VALUES
                (@AuctionId
                ,@BidderId
                ,@Bid
                ,@TimeOfBid
                ,@Id)"
            , new
            {
                Id = bidHistoryDTO.Id,
                BidderId = bidHistoryDTO.BidderId,
                Bid = bidHistoryDTO.Bid,
                TimeOfBid = bidHistoryDTO.TimeOfBid,
                AuctionId = bidHistoryDTO.AuctionId
            });
    }
}

public void PersistUpdateOf(IAggregateDataModel entity)
{
    throw new NotImplementedException();
}
}
}

```

Чтобы закончить реализацию репозитория, необходимо реализовать интерфейс единицы работы. Добавьте в папку `Infrastructure` новый класс с именем `UnitOfWork`, определение которого приводится в листинге 21.113.

Листинг 21.114. Реализация шаблона «Единица работы»

```

using System;
using System.Collections.Generic;
using System.Transactions;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class UnitOfWork : IUnitOfWork

```

```
{
    private Dictionary<IAggregateDataModel, IUnitOfWorkRepository>
        addedEntities;
    private Dictionary<IAggregateDataModel, IUnitOfWorkRepository>
        changedEntities;

    public UnitOfWork()
    {
        addedEntities = new Dictionary<IAggregateDataModel,
            IUnitOfWorkRepository>();
        changedEntities = new Dictionary<IAggregateDataModel,
            IUnitOfWorkRepository>();
    }

    public void RegisterAmended(IAggregateDataModel entity,
        IUnitOfWorkRepository unitOfWorkRepository)
    {
        if (!changedEntities.ContainsKey(entity))
        {
            changedEntities.Add(entity, unitOfWorkRepository);
        }
    }

    public void RegisterNew(IAggregateDataModel entity,
        IUnitOfWorkRepository unitOfWorkRepository)
    {
        if (!addedEntities.ContainsKey(entity))
        {
            addedEntities.Add(entity, unitOfWorkRepository);
        }
    };

    public void Clear()
    {
        addedEntities.Clear();
        changedEntities.Clear();
    }

    public void Commit()
    {
        using (TransactionScope scope = new TransactionScope())
        {
            foreach (IAggregateDataModel entity in this.addedEntities.Keys)
            {
                this.addedEntities[entity].PersistCreationOf(entity);
            }

            foreach (IAggregateDataModel entity in
                this.changedEntities.Keys)
            {
                this.changedEntities[entity].PersistUpdateOf(entity);
            }
        }
    }
}
```

```

        scope.Complete();
        Clear();
    }
}
}
}
}

```

Чтобы иметь возможность использовать класс `TransactionScope`, гарантирующий подтверждение сохранения в рамках атомарной транзакции, необходимо добавить ссылку на `System.Transactions`, как показано на рис. 21.23.

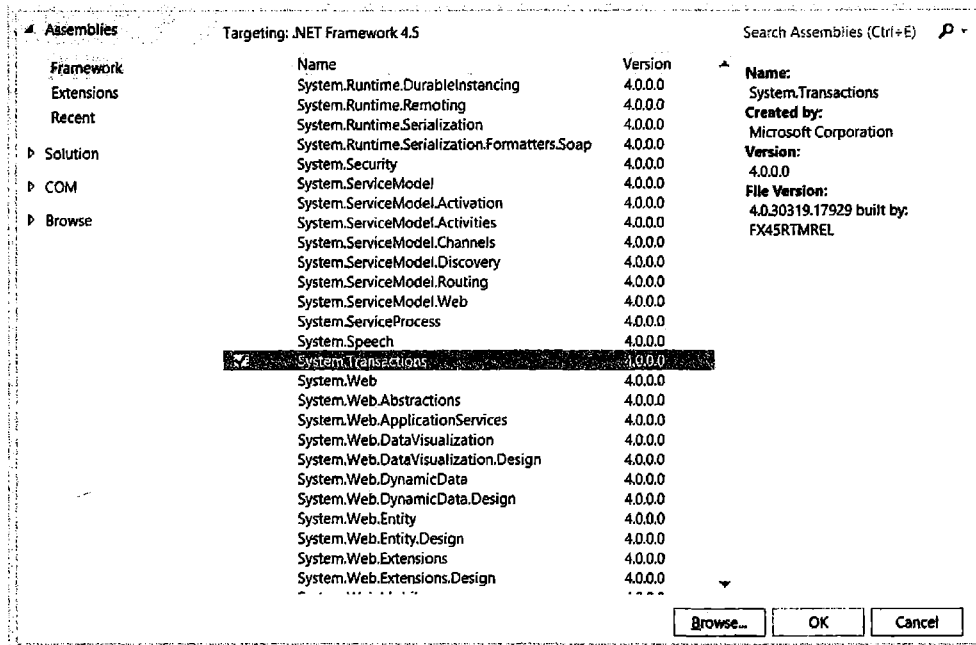


Рис. 21.23. Добавление ссылки на `System.Transactions`

Класс `UnitOfWork` использует три словаря для слежения за еще не сохраненными изменениями в предметных сущностях. Первый словарь предназначен для сущностей, добавляемых в хранилище, второй — для сущностей, подвергшихся изменениям, и третий — для сущностей, подлежащих удалению. Соответствующая реализация `IUnitOfWorkRepository` сохраняется в словаре со ссылкой на сущность в ключе и используется методом `Commit` для обращения к репозиторию, который содержит фактический код, сохраняющий сущность. Метод `Commit` обходит в цикле каждый словарь и вызывает соответствующие методы `IUnitOfWorkRepository`, передавая ссылку на сущность. Тело метода `Commit` завернуто в блок использования `TransactionScope`; это гарантирует, что никакая фактическая работа не будет выполнена до вызова метода `TransactionScope.Complete`. Если в ходе выполнения операций в `IUnitOfWorkRepository` возникнет исключение, все операции будут отменены и хранилище останется в исходном состоянии.

Конфигурация

И снова, внедрение всех зависимостей в прикладные службы будет выполняться с помощью контейнера StructureMap, который можно установить с помощью NuGet, как это было показано в примерах для NHibernate и RavenDB. Затем добавьте класс **Bootstrapper** (листинг 21.115) для настройки StructureMap.

Листинг 21.115. Класс Bootstrapper для настройки StructureMap

```
using System;
using StructureMap;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;

namespace DDDPPP.Chap21.MicroORM.Application
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                    .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();

                config.For<IUnitOfWork>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(x =>
                    {
                        var uow = new UnitOfWork();
                        return uow;
                    });
            });
        }
    }
}
```

Чтобы настроить строку подключения к базе данных, добавьте в файл `app.config` в проекте **Presentation** фрагмент разметки XML из листинга 21.116.

Листинг 21.116. Файл app.config со строкой подключения

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <connectionStrings>
```

```
<add name="AuctionDB"
      connectionString="Data Source=.\sqlexpress;
                        InitialCatalog=AuctionExample;
                        Integrated Security=True;
                        MultipleActiveResultSets=True"
      providerName="System.Data.SqlClient" />
</connectionStrings>
</configuration>
```

Теперь при запуске программы вы должны получить те же результаты, что и во всех предыдущих примерах.

Ключевые идеи

- Репозиторий играет роль передаточного звена между предметной моделью и моделью данных; он отображает предметную модель в хранилище.
- Шаблон «Репозиторий» — это процедурная граница, предохраняющая предметную модель от проникновения в нее инфраструктурных задач.
- Репозиторий — это контракт, а не уровень доступа к данным. Он четко определяет, какие операции доступны для каждого агрегата, то есть какие операции можно изменять, а какие удалять.
- Репозитории предназначены для использования внутри транзакций, а не для нужд составления отчетов. Для получения справочной информации используйте отдельные запросы.
- Репозиторий скрывает механизмы доступа к модели данных и поддерживает принцип «говори, а не спрашивай».
- Фреймворк ORM не является репозиторием. Репозиторий — это архитектурный шаблон, тогда как ORM подразумевает представление модели данных в виде объектной модели. Репозиторий может использовать ORM для передачи данных между предметной моделью и моделью данных.
- Используемый фреймворк сохранения влияет на конструкцию модели данных. Будьте прагматичными, не воюйте со своим фреймворком, стремясь к ненужной чистоте.
- Репозитории лучше всего подходят для ограниченных контекстов с богатой предметной моделью. Для простых ограниченных контекстов, не имеющих сложной предметной логики, используйте фреймворки сохранения напрямую.
- Управляя транзакциями, используйте шаблон «Единица работы». Единица работы запоминает, какие объекты удалялись, добавлялись или изменялись. Репозиторий отвечает за фактические операции с хранилищем, выполняемые в рамках транзакции, координация которой осуществляется единицей работы.
- Репозиторий должен отвечать только за изменение единственного агрегата; он не должен управлять транзакцией. Эта ответственность возлагается на реализацию единицы работы.

22

Регистрация событий

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в регистрацию событий и знакомство с проблемами, которые решает этот прием
- Примеры и рекомендации по созданию предметных моделей с поддержкой регистрации событий
- Как создать хранилище событий
- Примеры создания хранилищ событий поверх RavenDB и SQL Server
- Примеры использования хранилища событий, предложенного Грегом Янгом
- Как шаблон CQRS может помочь в организации регистрации событий
- Список компромиссов, который поможет понять, когда желательно использовать регистрацию событий, а когда следует избегать ее

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 22 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

В последние годы все большую популярность приобретает механизм хранения, который называют механизмом регистрации событий (event sourcing), потому что он дает дополнительные конкурентные преимущества и решает актуальные технические проблемы. Благодаря хранению полной хронологии действий, механизм регистрации событий помогает бизнесу глубже понять многие аспекты своих данных, включая особенности поведения клиентов. На основе этой хронологической информации можно сформировать новые и оригинальные предложения для производства продукта, выработки рыночных стратегий и других аспектов бизнеса. Использование регистрации событий позволяет определить состояние системы в любой момент времени и выяснить, как то или иное состояние было достигнуто. Для многих областей это совершенно уникальная возможность.

Многие современные системы хранят только текущее состояние предметной модели, не позволяя проанализировать изменение поведения во времени. Поэтому основой регистрации событий являются новые способы хранения данных и новые подходы к созданию предметных моделей, со своими достоинствами и недостатками. В этой главе будет представлено множество конкретных примеров таких сценариев. Вы даже сможете увидеть, как создается хранилище событий, и познакомитесь со специализированным хранилищем Event Store.

Практики предметно-ориентированного проектирования часто стараются объединить регистрацию событий с шаблоном CQRS, обеспечивающим улучшенную масштабируемость и производительность. Суть шаблона CQRS заключается в денормализации, но он настолько хорошо дополняет регистрацию событий, что вокруг этой комбинации сформировалось целое сообщество. Поэтому так важно устранить неоднозначные понятия и разобраться во всех деталях. Также важно сохранять трезвость ума в свете шумихи, генерируемой сообществом. Поэтому данная глава поможет также понять, когда регистрация событий может оказаться не самым эффективным выбором.

Чтобы вы были готовы к встрече с технически сложными примерами, эта глава начнется с разъяснения важности использования регистрации событий и сравнения с традиционной формой хранения предметных моделей, когда хранится только текущее состояние. Это разъяснение и все примеры в главе основаны на сценарии службы предоплаты, предлагаемой оператором сотовой связи. Воспользовавшись этой службой, клиенты пополняют свои счета с помощью ваучеров или карт предварительной оплаты. Такое пополнение позволяет им совершать телефонные звонки.

Ограничения на хранение состояния в виде моментального снимка

При хранении только текущего состояния предметной модели невозможно понять, как система достигла такого состояния. Как следствие, невозможно проанализировать хронологию событий, чтобы вскрыть новые возможности или понять причины неудач. Это можно показать на примере сценария, представленного отделом обслуживания клиентов оператора сотовой связи. Клиент выражает претензию, что его баланс опустел, хотя совсем недавно он пополнил счет и сделал лишь несколько коротких звонков. Клиент убежден, что у него должны были остаться средства на счету. Заглянув в базу данных, сотрудник отдела обслуживания видит информацию о счете, как показано в табл. 22.1.

Таблица 22.1. Состояние счета клиента, обслуживаемого по предварительной оплате

Идентификатор клиента	Доступно (минут)
123456789	0

Взглянув на счет клиента (см. табл. 22.1), сотрудник может лишь согласиться, что баланс опустел. Но он не сможет сказать, как было достигнуто такое состояние, не имея информации о недавних звонках и платежах. Подняв журналы и квитанции об оплате, отдел обслуживания восстановил хронологию списания средств со счета клиента, как показано в табл. 22.2.

Таблица 22.2. Хронология операций со счетом клиента

Прежде доступно (минут)	Действие	В настоящий момент доступно (минут)
0	Пополнение на \$10	20
20	Звонок длительностью 10 минут	10
10	Звонок длительностью 5 минут	0

ПРИМЕЧАНИЕ

Этот пример преднамеренно упрощен, чтобы подчеркнуть проблемы, которые могут возникать из-за хранения состояния в виде моментального снимка. Операторы сотовой связи могли бы хранить сведения обо всех звонках и фактах пополнения счета.

Исследовав табл. 22.2, можно заключить, что \$10 стоят 20 минут разговора по телефону. Однако баланс обнулится уже после двух разговоров — 10- и 5-минутного. Это ошибка в системе, но определить ее невозможно, имея перед глазами только текущее состояние, без хронологии действий клиента.

При хранении состояния в виде моментального снимка оператор сотовой связи также теряет возможность использовать хронологию действий клиента для повышения качества обслуживания. Он не сможет связать поведение пользователя с маркетинговыми акциями и выявить общие особенности поведения, которые можно было бы обратить себе в пользу или привлечь новые источники дохода.

В данном сценарии легко обосновать необходимость для оператора сотовой связи хотя бы рассмотреть возможность использования регистрации событий как способа хранения хронологии изменения счета клиента. Может быть, хранение состояния предметной модели в виде потока событий даст новые конкурентные преимущества и уменьшит количество жалоб клиентов.

Конкурентные преимущества от хранения состояния в виде потока событий

При сохранении важных событий в хронологическом порядке появляется возможность анализировать исторические данные. Текущее состояние модели можно получить, воспроизводя эти события. Но что особенно важно, можно получить не только текущее состояние; воспроизводя произвольную последовательность событий, можно выяснить, какое состояние имело место в тот или иной момент вре-

интересным является то, что он может объединить информацию о множестве счетов, чтобы выявить закономерности, связанные с определенными демографическими показателями. Такую информацию можно использовать для количественного анализа и принятия решений о направлениях развития продуктов или для постановки экспериментальных задач. Проекция — еще один важный механизм, позволяющий объединять события из множества потоков для выполнения подобных сложных временных запросов.

Проекции

Ограничением некоторых хранилищ события является сложность выполнения специализированных запросов к множеству потоков событий, которая в базах данных SQL легко преодолевается операцией соединения. Для решения этой проблемы хранилища событий используют понятие *проекций* (projections). Проекция — это запросы, отображающие множество входных потоков событий в один или несколько выходных потоков. Например, оператор сотовой связи может создать проекцию, отвечающую на такие вопросы: «сколько всего минут было использовано в определенный день демографическими группами во время определенного спортивного события?» и «увеличилось или уменьшилось число израсходованных минут в определенной демографической группе после того, как было сделано специальное предложение?». На рис. 22.2 показано, как можно находить ответы на подобные вопросы, объединяя множество потоков событий в проекции.

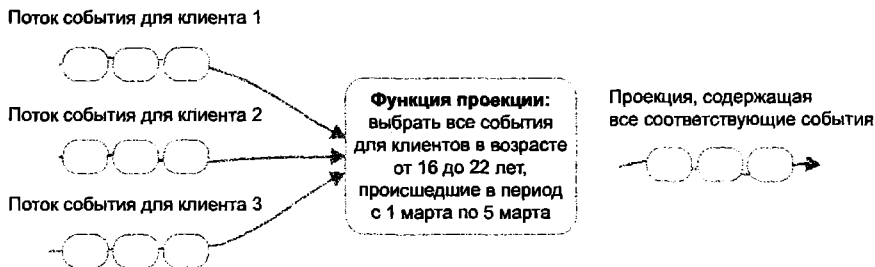


Рис. 22.2. Создание проекций из нескольких потоков событий

Как показано на рис. 22.2, на вход функции проекции подаются все потоки событий, каждый из которых представляет единственного клиента. Функция проекции может выполнить множество разных операций, включая сохранение состояния или порождение новых событий. Функция проекции на рис. 22.2 возвращает все события, имевшие место в период между 1 и 5 марта у клиентов в возрасте от 16 до 22 лет. Этот новый поток является проекцией. Объединив все необходимые события в единый поток, не составит никакого труда определить общее количество, среднее или любые другие характеристики.

Множество входных потоков можно также спроецировать в несколько выходных потоков. Как будет показано далее в этой главе, хранилище событий Грега Янга дает весьма широкие возможности и позволяет использовать всю мощь языка JavaScript в функциях проекций.

ПРИМЕЧАНИЕ

Дополнительные примеры использования проекций для создания отчетов на основе хранилищ событий можно найти в главе 26 «Запросы: предметная отчетность».

Моментальные снимки

Когда состояние сохраняется в виде событий, получаются потоки событий, которые могут расти очень быстро, что, в свою очередь, может привести к многократному увеличению времени, необходимого для воспроизведения событий. Чтобы избежать потерь производительности, сохраняются моментальные снимки событий. Как показано на рис. 22.3, моментальные снимки — это промежуточные точки в потоке событий, представляющие состояние, которое получается после воспроизведения всех предшествующих событий.

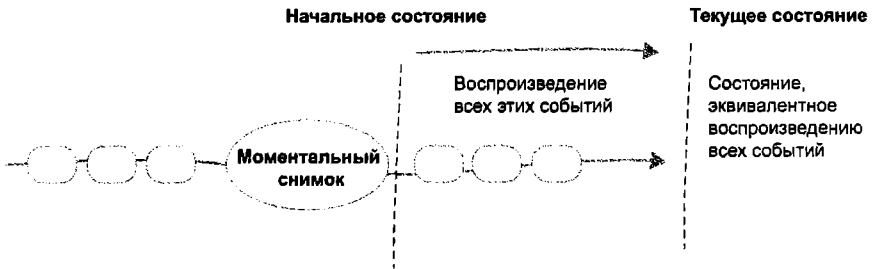


Рис. 22.3. Эффективное восстановление состояния с применением моментальных снимков

Когда приложению требуется определить текущее состояние агрегата по потоку событий, ему достаточно найти последний моментальный снимок и, используя его в роли начального состояния, воспроизвести только события, следующие за этим снимком в потоке.

ПРИМЕЧАНИЕ

На протяжении этой главы термины «поток событий» и «поток» будут использоваться взаимозаменяемо, так же как термины «предметное событие» и «событие». Это соответствует терминологии, используемой специалистами, на практике применяющими методики регистрации событий.

Агрегаты с поддержкой регистрации событий

Для поддержки регистрации событий агрегаты должны быть событийно-ориентированными. Также важно, чтобы они умели вычислять свое состояние, применяя последовательность событий. Практики DDD считают приятным побочным эффектом создания таких агрегатов тот факт, что они становятся более сосредоточенными на поведении и придают дополнительную выразительность предметным событиям. Еще одна выгода состоит в слабой зависимости от механизма хранения событий и простоте его реализации.

Структурирование

Создание агрегатов для модели с поддержкой регистрации событий имеет ряд отличительных особенностей. Из них наиболее важной является способность агрегата применить предметное событие и изменить свое состояние в соответствии с определенными бизнес-правилами. Во-вторых, должен поддерживаться список незафиксированных событий, чтобы впоследствии их можно было записать в хранилище событий. В-третьих, агрегат должен поддерживать запись своих версий, предоставлять возможность создавать моментальные снимки и восстанавливать свое состояние из них.

Добавление поддержки регистрации событий

Часто лучше всего определить базовый класс, как в листинге 22.1, инкапсулирующий общие функциональные возможности, которые будут наследовать все агрегаты.

Листинг 22.1. Базовый класс агрегатов, инкапсулирующий общие функциональные возможности

```
public abstract class EventSourcedAggregate : Entity
{
    public List<DomainEvent> Changes { get; private set; }
    public int Version { get; protected set; }

    public EventSourcedAggregate()
    {
        Changes = new List<DomainEvent>();
    }

    public abstract void Apply(DomainEvent changes);
}
```

Поле **Changes** — это коллекция незафиксированных событий, а **Version**, как можно догадаться, хранит номер версии агрегата, суть этого понятия объясняется чуть ниже. Метод **Apply()** принимает событие **DomainEvent**, которое агрегат должен обработать с применением бизнес-правил и изменить свое состояние.

В листинге 22.2 показана конкретная реализация агрегата (частично), представляющего счет клиента. Она демонстрирует, как можно увеличить выразительность метода **Apply()**.

Листинг 22.2. Агрегаты с поддержкой регистрации событий могут практически явно отражать бизнес-правила

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();
}
```

```
public PayAsYouGoAccount()
{ }

public override void Apply(DomainEvent @event)
{
    When((dynamic)@event);
    Version = Version ++;
}

private void When(CreditAdded creditAdded)
{
    _credit = _credit.Add(creditAdded.Credit);
}

private void When(PhoneCallCharged phoneCallCharged)
{
    _credit = _credit.Subtract(phoneCallCharged.CostOfCall);

    if (_freeCallAllowance != null)
        _freeCallAllowance.Subtract(phoneCallCharged.CoveredByAllowance);
}

private void When(AccountCreated accountCreated)
{
    Id = accountCreated.Id;
    _credit = accountCreated.Credit;
}

...
}
```

Реализация `Apply()` в листинге 22.2 использует немного «динамического волшебства», позволяющего придать высочайшую выразительность методам обработки событий. В листинге 22.2 такими методами являются методы `when()`. Каждая перегруженная версия метода `when()` читается почти как инструкция, отражающая применяемое бизнес-правило и описывающая порядок изменения состояния для каждого типа события. Каждый раз, когда вызывается метод `Apply()`, он также изменяет поле `Version`. Порой очень полезно определить, изменился ли агрегат, прежде чем выполнить операцию с ним. Версия агрегата — это порядковый номер последнего события, переданного методу `Apply()`.

Экспортирование выразительного предметно-ориентированного API

Метод `Apply()`, по сути, является деталью реализации, которая не будет использоваться за пределами агрегата. Он объявлен общедоступным только для того, чтобы дать возможность воссоздавать агрегат (объясняется чуть ниже). Службы за пределами агрегата все еще взаимодействуют посредством выразительных программных интерфейсов (API), ничего не зная о событийно-ориентированных особенностях. Это иллюстрирует метод `TopUp()` в измененной версии агрегата `PayAsYouGoAccount`, представленной в листинге 22.3.

Листинг 22.3. Измененный агрегат `PayAsYouGoAccount` с выразительным предметно-ориентированным API

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();

    public PayAsYouGoAccount()
    { }

    ...

    public void TopUp(Money credit, IClock clock)
    {
        if (_InclusiveMinutesOffer.IsSatisfiedBy(credit))
            Causes(new CreditSatisfiesFreeCallAllowanceOffer(
                this.Id, clock.Time(), _inclusiveMinutesOffer.FreeMinutes)
            );

        Causes(new CreditAdded(this.Id, credit));
    }

    private void Causes(DomainEvent @event)
    {
        Changes.Add(@event);
        Apply(@event);
    }

    ...
}
```

Листинг 22.3 показывает, как выглядит агрегат за пределами предметной модели с точки зрения прикладных служб. Им не известно, что агрегат использует методику регистрации событий; они видят лишь выразительные, высокоуровневые методы, такие как `TopUp()`, четко выражающие предметные понятия. С этой точки зрения ничто в агрегате не выдает поддержку регистрации событий.

Чтобы показать, что поддержка событий по большей части является лишь деталью реализации, а принцип взаимодействий прикладных служб с агрегатами остается прежним, в листинге 22.4 приводится реализация прикладной службы `TopUpCredit`. Попробуйте представить, что нет никаких событий, — тогда взаимодействия с агрегатом, не поддерживающим регистрацию событий, выглядели бы практически идентично.

Листинг 22.4. Взаимодействия с агрегатами из прикладных служб посредством высокоуровневого API

```
// прикладная служба
public class TopUpCredit
{
    private IPayAsYouGoAccountRepository _payAsYouGoAccountRepository;
```

```

private IDocumentSession _unitOfWork;
private IClock _clock;

public TopUpCredit(IPayAsYouGoAccountRepository payAsYouGoAccountRepository,
                  IClock clock)
{
    _payAsYouGoAccountRepository = payAsYouGoAccountRepository;
    _clock = clock;
}

public void Execute(Guid id, decimal amount)
{
    ...

    var account = _payAsYouGoAccountRepository.FindBy(id);
    var credit = new Money(amount);

    // .. выразительный, предметно-ориентированный API — ничто
    // даже не намекает на поддержку регистрации событий
    account.TopUp(credit, _clock);
    _payAsYouGoAccountRepository.Save(account);

    ...
}
}

```

Когда вызывается `TopUp()`, возникает предметное событие `CreditSatisfiedFreeCallAllowanceOffer` или `CreditAdded` — в зависимости от действия правила `InclusiveMinutesOfferBusiness`. Важно отметить, что в результате этих действий появляется новое событие. Это новое событие, после передачи в метод `Apply()`, должно вызвать изменение состояния агрегата. Но событие также должно быть сохранено, чтобы его можно было воспроизвести в процессе загрузки агрегата из его потока событий. Эту задачу решает метод `Causes()`, добавляя событие в `Changes` — коллекцию незафиксированных событий — и затем передавая его методу `Apply()`.

Добавление поддержки моментальных снимков

Осталось решить еще одну проблему, касающуюся агрегата, прежде чем переходить к обсуждению вопросов, связанных с хранением событий, — это возможность создания моментальных снимков. Агрегаты сами должны создавать моментальные снимки, потому что они содержат предметную логику, восстанавливающую состояние по предыдущим событиям. Соответственно было бы уместно добавить методы создания снимков и восстановления состояния из снимков непосредственно в агрегаты. Пример таких методов приводится в листинге 22.5.

Листинг 22.5. Добавление методов поддержки моментальных снимков в агрегаты

```

public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();
}

```

```

...

// перегруженный конструктор – восстанавливает агрегат из снимка
public PayAsYouGoAccount(PayAsYouGoAccountSnapshot snapshot)
{
    Version = snapshot.Version;
    _credit = new Money(snapshot.Credit);
}

public PayAsYouGoAccountSnapshot GetPayAsYouGoAccountSnapshot()
{
    return new PayAsYouGoAccountSnapshot
    {
        Version = Version,
        Credit = _credit.Amount
    };
}

...
}

```

Перегруженный конструктор в листинге 22.5 принимает моментальный снимок `PayAsYouGoSnapshot` и восстанавливает состояние агрегата, соответствующее этому состоянию. Как обсуждалось выше, эта оптимизация эквивалентна применению всех событий, возникавших перед созданием снимка. Для этого снимок должен в точности отражать состояние агрегата. Как можно видеть в листинге 22.5, выполнение этого требования гарантирует метод `GetPayAsYouGoAccountSnapshot()`, создающий снимок и заполняющий его соответствующими значениями. Определение `PayAsYouGoAccountSnapshot` приводится в листинге 22.6.

Листинг 22.6. Моментальный снимок, хранящий полное состояние агрегата

```

public class PayAsYouGoAccountSnapshot
{
    public int Version { get; set; }
    public decimal Credit { get; set; }
}

```

Сохранение и восстановление

Сохранение агрегатов, обладающих поддержкой регистрации событий, заключается в сохранении лишь незафиксированных изменений в хранилище событий. Аналогично, загрузка агрегата, или его *восстановление*, требует загрузить и воспроизвести все события, сохраненные прежде, возможно, с использованием моментального снимка для ускорения процесса. Как было показано в предыдущих примерах, для этих целей используются поле `Changes` и метод `Apply()` соответственно.

Создание репозитория с поддержкой регистрации событий

В листинге 22.7 приводится пример реализации репозитория, осуществляющего загрузку и сохранение агрегатов `PayAsYouGoAccount` (для начала без поддержки моментальных снимков).

Листинг 22.7. Репозиторий для сохранения и восстановления агрегатов с поддержкой регистрации событий

```
public class PayAsYouGoAccountRepository : IPayAsYouGoAccountRepository
{
    private readonly IEventStore _eventStore;

    public PayAsYouGoAccountRepository(IEventStore eventStore)
    {
        _eventStore = eventStore;
    }

    public void Add(PayAsYouGoAccount payAsYouGoAccount)
    {
        var streamName = StreamNameFor(payAsYouGoAccount.Id);
        _eventStore.CreateNewStream(streamName, payAsYouGoAccount.Changes);
    }

    public void Save(PayAsYouGoAccount payAsYouGoAccount)
    {
        var streamName = StreamNameFor(payAsYouGoAccount.Id);
        _eventStore.AppendEventsToStream(streamName, payAsYouGoAccount.Changes);
    }

    public PayAsYouGoAccount FindBy(Guid id)
    {
        var streamName = StreamNameFor(id);
        var fromEventNumber = 0;
        var toEventNumber = int.MaxValue ;

        var stream = _eventStore.GetStream(
            streamName, fromEventNumber, toEventNumber
        );

        var payAsYouGoAccount = new PayAsYouGoAccount();
        foreach(var @event in stream)
        {
            payAsYouGoAccount.Apply(@event);
        }

        return payAsYouGoAccount;
    }

    private string StreamNameFor(Guid id)
    {
        // отдельный поток для каждого агрегата: {AggregateType}-{AggregateId}
        return string.Format("{0}-{1}", typeof(PayAsYouGoAccount).Name, id);
    }
}
```

В листинге 22.7 показаны три основные операции, которые должны поддерживать репозитории: создание потоков, добавление в конец потоков и загрузка потоков. Создание потока событий заключается в создании нового потока с начальным набором событий, как показано в методе `Add()`. Аналогично, метод `Save()` добавляет

незафиксированные события из агрегата `PayAsYouGoAccount` в существующий поток. Наконец, метод `FindBy()` загружает `PayAsYouGoAccount` с заданным идентификатором.

Загрузка агрегатов выглядит сложнее, чем создание или изменение, потому что приходится решать, какие события должны загружаться из потока, с указанием высшего и низшего номера версий (это может пригодиться для восстановления агрегата в некотором предыдущем состоянии). После извлечения всех событий из хранилища они передаются методу `Apply()` агрегата, который воссоздает его состояние. Это единственный случай, когда внешняя служба должна вызывать `Apply()`.

Важно отметить в листинге 22.7, что имя потока событий имеет формат `{AggregateType}-{AggregateId}`, и тому есть веская причина. Наличие отдельного потока для каждого агрегата означает, что во время загрузки будут воспроизводиться только события для данного конкретного агрегата, а не для всех агрегатов данного типа. Это существенно улучшает производительность и создает предпосылки для лучшей масштабируемости. Кроме того, такой подход упрощает поддержку моментальных снимков.

Добавление поддержки снимков в операции сохранения и восстановления

Как уже говорилось выше, поддержка моментальных снимков помогает улучшить производительность и избежать необходимости загружать из потока всю последовательность событий. Моментальные снимки обычно создаются фоновым процессом, например службой Windows. Простейший пример фонового задания, создающего моментальные снимки агрегата `PayAsYouGoAccount` с использованием хранилища событий на основе `RavenDB`, показан в листинге 22.8.

Листинг 22.8. Пример фонового процесса, создающего моментальные снимки через определенные интервалы времени

```
public class PasAsYouGoAccountSnapshotJob
{
    private IDocumentStore _documentStore;

    public PasAsYouGoAccountSnapshotJob(IDocumentStore documentStore)
    {
        this._documentStore = documentStore;
    }

    public void Run()
    {
        while(true)
        {
            foreach (var id in GetIds())
            {
                using (var session = _documentStore.OpenSession())
                {
                    var repository = new PayAsYouGoAccountRepository(
                        new EventStore(session)
                    );
                }
            }
        }
    }
}
```

```

        var account = repository.FindBy(Guid.Parse(id));
        var snapshot = account.GetPayAsYouGoAccountSnapshot();
        repository.SaveSnapshot(snapshot, account);
    }
}

// создавать новый снимок для каждого агрегата каждые 12 часов
Thread.Sleep(TimeSpan.FromHours(12));
}
}

private IEnumerable<string> GetIds()
{
    using (var session = _documentStore.OpenSession())
    {
        return session.Query<EventStream>()
            .Select(x => x.Id)
            .ToList();
    }
}
}
}

```

Фоновое задание в листинге 22.8 раз в 12 часов обходит в цикле идентификаторы всех агрегатов и для каждого из них создает моментальный снимок. В действующей версии, возможно, желательно будет обрабатывать идентификаторы небольшими порциями, журналировать факты создания снимков и использовать не 12-часовой, а иной период, более оптимальный для вашего окружения. Также иногда необходимо создавать снимки по некоторому условию, например, когда число событий, происшедших с момента создания предыдущего снимка, превысит некоторое пороговое значение.

После создания и сохранения снимков можно оптимизировать стратегию загрузки, используя последний снимок в качестве начального состояния. В листинге 22.9 представлена дополненная реализация `PayAsYouGoRepository.FindBy()`, использующая снимки для оптимизации.

Листинг 22.9. Загрузка агрегата с использованием моментального снимка

```

public PayAsYouGoAccount FindBy(Guid id)
{
    var streamName = StreamNameFor(id);

    var fromEventNumber = 0;
    var toEventNumber = int.MaxValue ;

    var snapshot = _eventStore.GetLatestSnapshot<PayAsYouGoAccountSnapshot>(
        streamName
    );

    if (snapshot != null)
    {
        // загрузить только события, полученные после создания снимка
        fromEventNumber = snapshot.Version + 1;
    }
}

```

```

var stream = _eventStore.GetStream(streamName, fromEventNumber, toEventNumber);

PayAsYouGoAccount payAsYouGoAccount = null;
if (snapshot != null)
{
    payAsYouGoAccount = new PayAsYouGoAccount(snapshot);
}
else
{
    payAsYouGoAccount = new PayAsYouGoAccount();
}

foreach(var @event in stream)
{
    payAsYouGoAccount.Apply(@event);
}

return payAsYouGoAccount;
}

```

Здесь в метод `FindBy()` был добавлен вызов `_eventStore.GetLatestSnapshot()`. Если этот вызов вернет снимок, переменная `fromEventNumber` получит значение номера версии, следующего за номером версии снимка. Благодаря этому следующий вызов `_eventStore.GetStream()` вернет только события, возникшие после создания снимка. В листинге 22.9 можно также заметить другие изменения; если снимок имеется, агрегат `PayAsYouGoAccount` создается на основе этого снимка.

Решение проблем одновременного доступа

Иногда желательно, чтобы операция добавления события в поток терпела неудачу. Это обычная ситуация, когда несколько пользователей одновременно пытаются изменить состояние агрегата и требуется исключить возможность изменения, если агрегат в хранилище изменился уже после того, как новое изменение было произведено, но еще не было подтверждено. Возможно, вы уже знакомы с этим понятием: оптимистическая конкуренция ([http://technet.microsoft.com/en-us/library/aa213031\(v=sql.80\).aspx](http://technet.microsoft.com/en-us/library/aa213031(v=sql.80).aspx)).

Реализовать оптимистическое управление одновременным доступом (конкуренцией) в приложении с поддержкой регистрации событий можно, выполнив два дополнительных пункта. Во-первых, после загрузки агрегат должен сохранять номер версии, полученный в начале транзакции. Во-вторых, непосредственно перед добавлением в поток любого нового события нужно сравнить номер версии в хранилище с номером версии агрегата в момент начала транзакции. Этот процесс иллюстрируют листинги 22.10 и 22.11.

Листинг 22.10. Сохранение начального номера версии в агрегате

```

public class PayAsYouGoAccount : EventSourcedAggregate
{

```

```
// после инициализации не изменяется
public int InitialVersion { get; private set; }

public PayAsYouGoAccount(PayAsYouGoAccountSnapshot snapshot)
{
    Version = snapshot.Version;
    InitialVersion = snapshot.Version;
    _credit = new Money(snapshot.Credit);
}

...
}
```

Листинг 22.11. Проверка изменения агрегата сравнением номера версии из хранилища событий с номером версии агрегата

```
public class PayAsYouGoAccountRepository : IPayAsYouGoAccountRepository
{

    public void Save(PayAsYouGoAccount payAsYouGoAccount)
    {
        var streamName = StreamNameFor(payAsYouGoAccount.Id);
        var expectedVersion = GetExpectedVersion(payAsYouGoAccount.InitialVersion);
        _eventStore.AppendEventsToStream(streamName, payAsYouGoAccount.Changes,
                                         expectedVersion);
    }

    private int? GetExpectedVersion(int expectedVersion)
    {
        if (expectedVersion == 0)
        {
            // когда агрегат сохраняется впервые, он не имеет ожидаемой версии
            return null;
        }
        else
        {
            return expectedVersion;
        }
    }

    ...
}
```

Листинг 22.10 содержит версию агрегата `PayAsYouGoAccount`, дополненную свойством `InitialVersion`, которое представляет версию последнего события, сохраненного перед загрузкой агрегата. Это значение затем передается в метод `AppendEventsToStream()` хранилища событий, как показано в листинге 22.11, а реализация хранилища событий может проверить, является ли данный номер версии последним. Полную реализацию этой особенности вы увидите далее в этой главе, где будет представлена реализация хранилища событий.

Тестирование

Модульное тестирование агрегатов с поддержкой регистрации событий подразумевает проверку появления событий. Делается это путем проверки коллекции незафиксированных событий, принадлежащих агрегату. Пример такого теста, проверяющего метод `PayAsYouGoAccount.TopUp()`, приводится в листинге 22.12.

Листинг 22.12. Модульное тестирование агрегатов с поддержкой регистрации событий путем проверки появления событий

```
[TestClass]
public class PayAsYouGoAccount_Tests
{
    static PayAsYouGoAccount _account;
    static Money _fiveDollars = new Money(5);
    static PayAsYouGoInclusiveMinutesOffer _free90MinsWith10DollarTopUp =
        new PayAsYouGoInclusiveMinutesOffer(
            new Money(10), new Minutes(90)
        );

    [ClassInitialize] // выполняется первым
    public static void
    When_applying_a_top_up_that_does_not_qualify_for_inclusive_minutes(
        TestContext ctx)
    {
        _account = new PayAsYouGoAccount(Guid.NewGuid(), new Money(0));
        // удалить событие AccountCreated, не имеющее отношения к этому тесту
        _account.Changes.Clear();

        _account.AddInclusiveMinutesOffer(_free90MinsWith10DollarTopUp);
        // платеж $5 меньше порога $10, позволяющего получить бесплатные минуты
        _account.TopUp(_fiveDollars, new SystemClock());
    }

    [TestMethod]
    public void
    The_account_will_be_credited_with_the_top_up_amount_but_no_free_minutes()
    {
        var lastEvent = _account.Changes.SingleOrDefault() as CreditAdded;
        Assert.IsNotNull(lastEvent);
        Assert.AreEqual(_fiveDollars, lastEvent.Credit);
        Assert.AreEqual(5, _account.GetPayAsYouGoAccountSnapshot().Credit);
    }
}
```

Если для клиента с учетной записью `PayAsYouGoAccount` действует предложение о предоставлении бесплатных минут и сумма платежа превышает установленное пороговое значение, клиент получает бесплатные минуты вдобавок к оплаченным. Тест в листинге 22.12 проверяет ситуацию, когда сумма платежа оказывается ниже порога и бесплатные минуты не должны даваться. Делается это проверкой наличия единственного события `CreditAdded` в списке незафиксированных событий агрегата и увеличения числа минут только на число оплаченных минут. Эти проверки осуществляются вызовами `Assert.Equal()` в конце листинга 22.12.

Листинг 22.12 представляет типичный тест для агрегатов с поддержкой регистрации событий; взаимодействия осуществляются с использованием выразительных, высокоуровневых программных интерфейсов (API), таких как `TopUp()`. Затем проверяется факт добавления события в список незафиксированных событий и соответствующее изменение состояния. Такой прием часто используется и в высокоуровневых интеграционных тестах, и в низкоуровневых модульных. Главное отличие — использование подставных реализаций вместо внешних объектов и служб или их действующих реализаций.

Создание хранилища событий

При создании приложения, поддерживающего регистрацию событий, можно использовать готовое специализированное хранилище событий или создать собственное. Под созданием собственного хранилища здесь подразумевается лишь использование имеющихся инструментов иным способом. То есть в этом разделе вы увидите, как можно использовать RavenDB и SQL Server в качестве основы хранилища событий. Создание собственного хранилища — отличный способ разобраться в основных понятиях.

Вы уже встречались с интерфейсом `IEventStore` ранее, в реализации `PayAsYouGoAccountRepository` в листинге 22.11. Этот интерфейс будет реализован хранилищем событий, представленным в этом разделе. Полное определение `IEventStore` показано в листинге 22.13.

Листинг 22.13. Интерфейс `IEventStore`

```
public interface IEventStore
{
    void CreateNewStream(string streamName, IEnumerable<DomainEvent> domainEvents);

    void AppendEventsToStream(string streamName, IEnumerable<DomainEvent>
                             domainEvents, int? expectedVersion);

    IEnumerable<DomainEvent> GetStream(string streamName, int fromVersion,
                                     Int toVersion);

    void AddSnapshot<T>(string streamName, T snapshot);

    T GetLatestSnapshot<T>(string streamName) where T: class;
}
```

ВНИМАНИЕ

Абстрагирование доступа к данным требует идти на компромиссы. Абстракция дает возможность менять технологии, подставляя альтернативные реализации. С другой стороны, абстракции определяют строгий интерфейс, что может препятствовать использованию некоторых дополнительных особенностей технологий доступа к данным. Зная, какую цену придется заплатить и насколько высока вероятность смены технологии в будущем, вы сможете принять правильное решение в отношении использования абстракций. Далее в этой главе обсуждается, почему регистрация событий способствует независимости от типа хранилища.

Проектирование структуры хранилища

Проектирование, или выбор, структуры хранилища — одна из самых сложных задач, возникающих при создании хранилища событий. Выбранная технология должна как минимум поддерживать создание потоков событий, добавление событий в конец потоков и извлечение событий в том же порядке. Также, весьма вероятно, вам понадобится поддержка моментальных снимков.

В этом примере структура хранилища основана на документах трех типов: `EventStream`, `EventWrapper` и `SnapshotWrapper`. `EventStream` представляет поток событий, но содержит только метаданные, такие как `Id` и `Version`. `EventWrapper` обертывает и представляет отдельное предметное событие, содержащее все сопутствующие данные плюс метаданные о потоке, которому оно принадлежит. `SnapshotWrapper` представляет единственный моментальный снимок. Как вариант, можно было бы определить единственный документ, включающий метаданные, события и моментальные снимки для каждого агрегата. Вы можете свободно проектировать и совершенствовать собственный формат по своему усмотрению.

RavenDB — документоориентированная база данных, в которой каждый документ хранится в формате JSON (JavaScript Object Notation — форма записи объектов JavaScript). Клиентская библиотека Raven для C# способна автоматически преобразовывать классы POCO (Plain Old C# Object — старый добрый объект C#) в разметку JSON. Это означает, что в коде достаточно просто объявить три типа документов, образующих основу функциональных возможностей хранилища событий. Определения `EventStream`, `EventWrapper` и `SnapshotWrapper` показаны в листингах с 22.14 по 22.16 соответственно.

Листинг 22.14. EventStream

```
public class EventStream
{
    public string Id { get; private set; } // тип агрегата + id
    public int Version {get; private set;}

    private EventStream() { }

    public EventStream(string id)
    {
        Id = id;
        Version = 0;
    }

    public EventWrapper RegisterEvent(DomainEvent @event)
    {
        Version ++;

        return new EventWrapper(@event, Version, Id);
    }
}
```

Листинг 22.15. EventWrapper

```
public class EventWrapper
{
    public string Id { get; private set; }
```



```
public DomainEvent Event { get; private set; }
public string EventStreamId { get; private set; }
public int EventNumber { get; private set; }

public EventWrapper(DomainEvent @event, int eventNumber, string streamStateId)
{
    Event = @event;
    EventNumber = eventNumber;
    EventStreamId = streamStateId;
    Id = string.Format("{0}-{1}", streamStateId, EventNumber);
}
}
```

Листинг 22.16. SnapshotWrapper

```
public class SnapshotWrapper
{
    public string StreamName { get; set; }

    public Object Snapshot { get; set; }

    public DateTime Created { get; set; }
}
```

Создание потоков событий

После выбора типа документа для представления потока событий, чтобы создать экземпляр потока, нужно создать документ этого типа. Это демонстрирует начальная реализация хранилища событий, поддерживающая создание потоков, как показано в листинге 22.17.

Листинг 22.17. Реализация хранилища, создающего потоки событий

```
public class EventStore : IEventStore
{
    private readonly IDocumentSession _documentSession;

    public EventStore(IDocumentSession documentSession)
    {
        _documentSession = documentSession;
    }

    public void CreateNewStream(string streamName,
        IEnumerable<DomainEvent> domainEvents)
    {
        var eventStream = new EventStream(streamName);
        _documentSession.Store(eventStream);

        AppendEventsToStream(streamName, domainEvents); // см. следующий листинг
    }
}
```

Используемая здесь технология хранения RavenDB представлена ее интерфейсом IDocumentSession. В листинге 22.17 можно видеть, что реализа-

ция `CreateNewStream()` использует `IDocumentSession` для сохранения объекта `EventStream`. Как упоминалось выше, `RavenDB` автоматически преобразует этот объект `EventStream` в формат `JSON` и создает новый документ для него. В этот момент, когда `RavenDB` создает документ `EventStream`, фактически создается новый поток событий. Однако он пока не содержит никаких событий.

Добавление событий в потоки

Следующей задачей после создания потоков событий является реализация добавления в них собственно событий. В этом примере каждое отдельное событие представлено классом `EventWrapper`, обертывающим предметное событие и содержащим метаданные, помогающие связать событие с определенным потоком при сохранении. Реализация `EventStore`, дополненная поддержкой добавления объектов `EventWrapper`, показана в листинге 22.18.

Листинг 22.18. Поддержка добавления событий в потоки

```
public class EventStore : IEventStore
{
    ...
    public void AppendEventsToStream(string streamName,
        IEnumerable<DomainEvent> domainEvents, int? expectedVersion = null)
    {
        var stream = _documentSession.Load<EventStream>(streamName);

        foreach (var @event in domainEvents)
        {
            _documentSession.Store(stream.RegisterEvent(@event));
        }
    }
}
```

`AppendEventsToStream()` сохраняет каждый объект `EventWrapper` как уникальный документ. Но это лишь деталь реализации; все клиенты, вызывающие `AppendEventsToStream()`, знают, что событие будет добавлено в конец соответствующего потока. Но в связи с тем что каждое событие является уникальным документом, необходим способ извлекать все события, принадлежащие заданному потоку. Класс `EventWrapper` хранит необходимые для этого метаданные: идентификатор потока и номер версии события. Эти значения заполняются вызовом `EventStream.RegisterEvent()`, как было показано выше, в листинге 22.14. Как вы увидите в следующем примере, извлечение событий из потока фактически сводится к поиску событий со значением `EventStreamId`, совпадающим с идентификатором заданного потока.

Извлечение потоков событий

В процессе разработки функциональности хранилища событий в нее должна быть как минимум включена возможность извлечения всех событий, принадлежащих заданному потоку. Также, для поддержки моментальных снимков, нужна возмож-

ность извлекать события по номеру версии или идентификатору. Дополненная реализация `EventStore`, поддерживающая эти возможности, показана в листинге 22.19.

Листинг 22.19. Поддержка загрузки событий из потока

```
public class EventStore : IEventStore
{
    ...

    public IEnumerable<DomainEvent> GetStream(string streamName,
        int fromVersion, int toVersion)
    {
        // извлечь события, начиная с указанной версии
        var eventWrappers = (from stream in _documentSession.Query<EventWrapper>()
            .Customize(x => x.WaitForNonStaleResultsAsOfNow())
            where stream.EventStreamId.Equals(streamName)
            && stream.EventNumber <= toVersion
            && stream.EventNumber >= fromVersion
            orderby stream.EventNumber
            select stream).ToList();

        if (eventWrappers.Count() == 0) return null;

        var events = new List<DomainEvent>();

        foreach (var @event in eventWrappers)
        {
            events.Add(@event.Event);
        }

        return events;
    }
    ...
}
```

Как обсуждалось выше, для представления потоков событий каждое событие в этом примере сохраняется как уникальный документ `EventWrapper`, содержащий идентификатор соответствующего потока. В листинге 22.19 можно видеть, что эта особенность поддерживается предложением `where`, указывающим, что выбираться должны только экземпляры `EventWrapper` с указанным идентификатором потока.

Для ограничения множества событий, загружаемых из потока, в запросе используются параметры `fromVersion` и `toVersion`. Они определяют первое и последнее событие, извлекаемое из потока, соответственно. Как будет показано в следующем примере, эта особенность пригодится для реализации поддержки моментальных снимков.

Добавление поддержки моментальных снимков

Для поддержки моментальных снимков необходимо предусмотреть способ их хранения. В данном примере для этой цели используется тип `SnapshotWrapper`, обертывающий моментальные снимки и добавляющий в них дополнительные

метаданные — идентификатор соответствующего потока и дату добавления. В листинге 22.20 демонстрируется версия EventStore, дополненная возможностью сохранения моментальных снимков.

Листинг 22.20. Поддержка сохранения моментальных снимков

```
public class EventStore : IEventStore
{
    ...

    public void AddSnapshot<T>(string streamName, T snapshot)
    {
        var wrapper = new SnapshotWrapper
        {
            StreamName = streamName,
            Snapshot = snapshot,
            Created = DateTime.Now
        };

        _documentSession.Store(snapshot);
    }

    ...
}
```

AddSnapshot<T>() позволяет клиентам сохранять моментальные снимки для заданного потока, обертывая их в документы SnapshotWrapper, содержащие идентификатор соответствующего потока и время сохранения. Как показано в листинге 22.21, эти метаданные позволяют клиентам EventStore получить последний снимок для любого потока, просто указав идентификатор этого потока.

Листинг 22.21. Поиск последнего снимка по идентификатору потока

```
public class EventStore : IEventStore
{
    ...

    public void AddSnapshot<T>(string streamName, T snapshot)
    {
        var wrapper = new SnapshotWrapper
        {
            StreamName = streamName,
            Snapshot = snapshot,
            Created = DateTime.Now
        };

        _documentSession.Store(snapshot);
    }

    public T GetLatestSnapshot<T>(string streamName) where T: class
    {
        var latestSnapshot = _documentSession.Query<SnapshotWrapper>()
            .Customize(x => x.WaitForNonStaleResultsAsOfNow())
```

```
        .Where(x => x.StreamName == streamName)
        .OrderByDescending(x => x.Created)
        .FirstOrDefault();

    if (latestSnapshot == null)
    {
        return null;
    }
    else
    {
        // развернуть снимок – вернуть клиенту то,
        // что он когда-то передавал в AddSnapshot
        return (T)latestSnapshot.Snapshot;
    }
}
```

Хотелось бы надеяться, что, исследовав код в листинге 22.21, вы заметите преимущества использования `SnapshotWrapper`. Поля `StreamName` и `Created` из этого документа используются для конструирования запроса, отыскивающего последний сохраненный моментальный снимок для указанного потока. В сочетании с `AddSnapshot<T>()` образуется ясный API для сохранения снимков любого типа и их извлечения в последующем. Но это лишь одно из множества возможных решений; вы можете попробовать поискать альтернативное решение, лучше соответствующее вашим потребностям. Творческий подход к решению задачи можно только приветствовать.

Управление одновременным доступом

Многие современные приложения являются многопользовательскими в том смысле, что позволяют нескольким пользователям одновременно просматривать и изменять одни и те же фрагменты данных. Если один пользователь пытается изменить некоторые данные, но в это же время другой пользователь изменил их, не согласуя свои действия с первым, изменения, внесенные одним из пользователей, иногда могут быть потеряны. Как уже упоминалось, это оптимистическая форма управления одновременным доступом. Типичный способ реализации оптимистической формы управления одновременным доступом к хранилищу событий заключается в проверке совпадения номера версии последнего сохраненного события с номером версии события, описывающего новые изменения.

Хранение ожидаемого номера версии на прикладном уровне было показано выше, в листинге 22.11, где в агрегат было добавлено свойство `InitialVersion`, которое затем передавалось в хранилище событий всякий раз, когда в поток добавлялось новое событие. В листинге 22.22 показано, как можно изменить метод `EventStore.AppendEventsToStream()`, использующий этот номер версии, чтобы предотвратить сохранение любых событий с номерами версий, не совпадающими с номером версии в последнем хранимом событии. Такое последнее событие (или события) могло быть добавлено другим пользователем уже после начала транзакции, и потому текущий пользователь мог просто не знать об этих изменениях.

Листинг 22.22. Добавление проверки оптимистической конкуренции при сохранении событий

```
public void AppendEventsToStream(string streamName,
    IEnumerable<DomainEvent> domainEvents, int? expectedVersion = null)
{
    var stream = _documentSession.Load<EventStream>(streamName);

    if (expectedVersion != null)
    {
        CheckForConcurrencyError(expectedVersion, stream);
    }

    foreach (var @event in domainEvents)
    {
        _documentSession.Store(stream.RegisterEvent(@event));
    }
}

private static void CheckForConcurrencyError(int? expectedVersion,
    EventStream stream)
{
    var lastUpdatedVersion = stream.Version;
    if (lastUpdatedVersion != expectedVersion)
    {
        var error = string.Format("Expected: {0}. Found: {1}",
            expectedVersion,
            lastUpdatedVersion);
        throw new OptimsticConcurrencyException(error);
    }
}
```

Если в вызов `AppendEventsToStream` передан номер ожидаемой версии, перед добавлением нового события `EventStore` проверит совпадение этого номера с номером версии последнего хранящегося события. Это видно в листинге 22.22. В частности, приватный метод `CheckForConcurrencyError()` осуществляет сравнение и возбуждает исключение `OptimisticConcurrencyException`, если номера версий не совпадают.

К сожалению, решение в листинге 22.22 не имеет достаточного уровня надежности при использовании вместе с `RavenDB`, потому что сохраняется вероятность «гонки» (race condition). Если взглянуть внимательнее, можно заметить, что между завершением `CheckForConcurrencyError()` и вызовом `_documentSession.Store()` остается короткий промежуток времени. Представьте, что получится, если новое событие будет добавлено в поток как раз в этот промежуток. Оба события будут добавлены в поток, несмотря на попытку `CheckForConcurrencyError()` предотвратить это. К счастью, есть возможность настроить в `RavenDB` поддержку оптимистической формы управления одновременным доступом, то есть предотвратить возможность сохранения события, если поток изменился уже после появления события.

Какую бы технологию вы ни выбрали для создания собственного хранилища событий, вам наверняка придется учитывать вероятность состояния «гонки», если вы

собираетесь поддерживать оптимистическую форму управления одновременным доступом. Для этого вы чаще всего будете использовать блокировку потока и проверку идентификатора последнего зафиксированного события. Чтобы заблокировать поток событий в RavenDB, сначала нужно включить поддержку оптимистической формы управления одновременным доступом, как показано в листинге 22.23.

Листинг 22.23. Включение поддержки оптимистической формы управления одновременным доступом в RavenDB

```
public static class Bootstrapper
{
    public static void Startup()
    {
        var documentStore = new DocumentStore
        {
            ConnectionStringName = "RavenDB"
        }.Initialize();

        documentStore
            .DatabaseCommands
            .EnsureDatabaseExists("EventSourcingExample");

        ObjectFactory.Initialize(config =>
        {
            // зарегистрировать здесь другие зависимости
            config.For<IDocumentStore>().Use(documentStore);
            config.For<IDocumentSession>()
                .HybridHttpOrThreadLocalScoped()
                .Use(x =>
                {
                    var store = x.GetInstance<IDocumentStore>();
                    var session = store.OpenSession();
                    session.Advanced.UseOptimisticConcurrency = true;
                    return session;
                });
        });
    }
}
```

RavenDB реализует поддержку оптимистической формы управления одновременным доступом. В листинге 22.23 показано, что для ее включения достаточно присвоить значение `true` свойству `session.Advanced.UseOptimisticConcurrency`. Также листинг 22.23 показывает, что настройку можно выполнить на уровне служб (подробнее об этом рассказывается в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования»), где осуществляется управление жизненным циклом объектов, обычно внутри контейнера. (`ObjectFactory` в листинге 22.23 — это IoC-контейнер.) По сути, программный код в листинге 22.23 гарантирует, что каждый новый веб-запрос или каждый поток выполнения получит собственный документ `session` с включенной оптимистической формой управления одновременным доступом. Очень важно выделять соб-

ственный сеанс каждому потоку выполнения или веб-запросу, чтобы иметь возможность откатить сразу все изменения, произведенные в сеансе.

После включения оптимистической формы управления одновременным доступом RavenDB будет прерывать транзакции и возбуждать исключение `ConcurrencyExceptions`, если сохраняемое событие было изменено в другом экземпляре `IDocumentSession` уже после создания текущего экземпляра `IDocumentSession`. Листинг 22.24 иллюстрирует, как это относится к `EventStore`, демонстрируя полную реализацию сценария использования в виде прикладной службы `TopUpCredit`.

Листинг 22.24. Обработка ошибок одновременного доступа в RavenDB

```
public class TopUpCredit
{
    private IPayAsYouGoAccountRepository _payAsYouGoAccountRepository;
    private IDocumentSession _unitOfWork;
    private IClock _clock;

    public TopUpCredit(IPayAsYouGoAccountRepository payAsYouGoAccountRepository,
        IDocumentSession unitOfWork, IClock clock)
    {
        _payAsYouGoAccountRepository = payAsYouGoAccountRepository;
        _unitOfWork = unitOfWork;
        _clock = clock;
    }

    public void Execute(Guid id, decimal amount)
    {
        try
        {
            var account = _payAsYouGoAccountRepository.FindBy(id);

            var credit = new Money(amount);

            account.TopUp(credit, _clock);

            _payAsYouGoAccountRepository.Save(account);

            _unitOfWork.SaveChanges();
        }
        catch (ConcurrencyException ex)
        {
            _unitOfWork.Advanced.Clear();

            // любые дополнительные операции обработки ошибок – повторная
            // отправка сообщений, добавления сообщений в очередь ошибок,
            // и т. д.

            throw ex;
        }
    }
}
```


В начале обработки веб-запроса создается экземпляр `IDocumentSession` и передается прикладной службе `TopUpCredit`, представленной в листинге 22.24. `TopUpCredit` затем находит соответствующий счет и зачисляет указанную сумму, вызывая события, которые должны быть добавлены в список незафиксированных событий агрегата. Когда вызывается метод `Save()` репозитория, эти события ставятся в очередь для сохранения в RavenDB. Наконец, когда вызывается `SaveChanges()` экземпляра `IDocumentSession (_unitOfWork)`, RavenDB фиксирует изменения и записывает их на диск. Но если RavenDB заметит, что поток, соответствующий этим событиям, был изменен другим экземпляром `IDocumentSession` уже после вызова `TopUpCredit`, транзакция будет прервана и вместо фиксации изменений будет возбуждено исключение `ConcurrencyException`, обработка которого предусмотрена в листинге 22.24.

Хранилище событий на основе SQL Server

SQL Server — еще одна распространенная база данных, которую можно использовать для организации хранилища событий. Существует несколько открытых проектов, которые помогут получить представление об особенностях реализации. Наибольшей популярностью среди них пользуются Ncqrs (<https://github.com/ncqrs/ncqrs/blob/master/Framework/src/Ncqrs/Eventing/Storage/SQL/MsSqlServerEventStore.cs>) и NEventStore (<https://github.com/NEventStore/NEventStore.Persistence.SQL/blob/master/src/NEventStore.Persistence.Sql/SqlPersistenceEngine.cs>). В следующем коротком разделе мы будем использовать Ncqrs в качестве учебного примера.

Выбор схемы

Выбирая базу данных SQL для использования в качестве хранилища событий и потоков, важно особое внимание уделить схеме данных. Те, кто уже имеет опыт создания хранилищ событий на основе баз данных SQL, предлагают использовать минималистский подход и хранить данные о событиях в виде больших двоичных объектов (blobs) с разметкой XML или JSON. Рекомендуется использовать универсальную схему данных, не зависящую от предметной области, как показано в листинге 22.25.

Листинг 22.25. Универсальная схема для хранилища событий на основе SQL Server

Table Events:

```
Id [uniqueidentifier] NOT NULL,  
TimeStamp [datetime] NOT NULL,  
  
Name [varchar](max) NOT NULL,  
Version [varchar](max) NOT NULL,  
  
EventSourceId [uniqueidentifier] NOT NULL,  
Sequence [bigint],  
  
Data [nvarchar](max) NOT NULL
```

Table EventSources:

```
Id [uniqueidentifier] NOT NULL,
Type [nvarchar](255) NOT NULL,
Version [int] NOT NULL
```

Table Snapshots:

```
EventSourceId [uniqueidentifier] NOT NULL,
Version [bigint] NULL,
TimeStamp [datetime] NOT NULL,
Type [varchar](255) NOT NULL,
Data [varbinary](max) NOT NULL
```

Одно из отличий от хранилища на основе RavenDB, рассмотренного выше, состоит в использовании здесь значений `[uniqueidentifier]`, также известных как GUID (Globally Unique Identifier — глобально-уникальный идентификатор), в качестве идентификаторов событий и потоков (которые в листинге 22.25 называются источниками событий — EventSources). В своем приложении вы можете использовать любой другой тип. Кроме этого отличия, можно заметить сходство с решением на основе RavenDB: каждое событие сохраняется в уникальной записи, а понятие потока является логическим — принадлежность событий к потоку определяется идентификатором потока.

Создание потока

Большая часть реализации хранилища событий Ncqrs определяется схемой. Например, создание нового потока заключается в простом создании новой записи в таблице EventSources. Это можно видеть в листинге 22.26. Отметим, что Ncqrs использует термин *event source* (источник событий), вместо которого везде в этой главе мы употребляем термин *event stream* (поток событий).

Листинг 22.26. Логика создания потока в Ncqrs

```
private static void AddEventSource(Guid eventSourceId, Type eventSourceType,
    long initialVersion, SqlTransaction transaction)
{
    using (var command =
        new SqlCommand(Queries.InsertNewProviderQuery,
            transaction.Connection))
    {
        command.Transaction = transaction;
        command.Parameters.AddWithValue("Id", eventSourceId);
        command.Parameters.AddWithValue("Type", eventSourceType.ToString());
        command.Parameters.AddWithValue("Version", initialVersion);
        command.ExecuteNonQuery();
    }
}

internal static class Queries
{

```

```

public const String InsertNewProviderQuery =
    "INSERT INTO [EventSources](Id, Type, Version) VALUES (@Id, @Type, @Version)";

...
}

```

Сохранение событий

Схема также определяет порядок сохранения событий. Листинг 22.27 демонстрирует это, показывая, как добавляется каждое событие в новую запись в таблице Events со ссылкой на соответствующий поток событий.

Листинг 22.27. Логика сохранения событий в Ncqrs

```

private void SaveEvents(IEnumerable<UncommittedEvent> uncommittedEvents,
    SqlTransaction transaction)
{
    foreach (var sourcedEvent in uncommittedEvents)
    {
        SaveEvent(sourcedEvent, transaction);
    }
}

private void SaveEvent(UncommittedEvent uncommittedEvent, SqlTransaction transaction)
{
    string eventName;
    var document = _formatter.Serialize(uncommittedEvent.Payload, out eventName);
    var storedEvent = new StoredEvent<JObject>{
        uncommittedEvent.EventIdentifier,
        uncommittedEvent.EventTimeStamp, eventName,
        uncommittedEvent.EventVersion,
        uncommittedEvent.EventSourceId,
        uncommittedEvent.EventSequence, document
    };

    var raw = _translator.TranslateToRaw(storedEvent);

    using (var command = new SqlCommand(
        Queries.InsertNewEventQuery, transaction.Connection))
    {
        command.Transaction = transaction;
        command.Parameters.AddWithValue("EventId", raw.EventIdentifier);
        command.Parameters.AddWithValue("TimeStamp", raw.EventTimeStamp);
        command.Parameters.AddWithValue("EventSourceId", raw.EventSourceId);
        command.Parameters.AddWithValue("Name", raw.EventName);
        command.Parameters.AddWithValue("Version", raw.EventVersion.ToString());
        command.Parameters.AddWithValue("Sequence", raw.EventSequence);
        command.Parameters.AddWithValue("Data", raw.Data);
        command.ExecuteNonQuery();
    }
}

```

```
internal static class Queries
{
    ...

    public const String InsertNewEventQuery =
        "INSERT INTO [Events] " +
        "    ([Id], [EventSourceId], [Name], [Version], " +
        "    [Data], [Sequence], [TimeStamp]) " +
        " VALUES "
        "    (@EventId, @EventSourceId, @Name, @Version, " +
        "    @Data, @Sequence, @TimeStamp)";

    ...
}
```

Загрузка событий из потока

Метод `ReadFrom()` в листинге 22.28 основан на примере из `Ncqrs` и реализует чтение событий из потока. Он является аналогом метода `IEventStore.GetStream()`, показанного ранее в этой главе, и имеет схожую логику. Метод возвращает все события, ссылающиеся на указанный поток (`id`) и попадающие в диапазон `minVersion` и `maxVersion`.

Листинг 22.28. Извлечение событий в `Ncqrs` из хранилища событий на основе `SQL Server`

```
public CommittedEventStream ReadFrom(Guid id, long minVersion, long maxVersion)
{
    var events = new List<CommittedEvent>();

    using (var connection = new SqlConnection(_connectionString))
    {
        using (var command =
            new SqlCommand(Queries.SelectAllEventsQuery, connection))
        {
            command.Parameters.AddWithValue("EventSourceId", id);
            command.Parameters.AddWithValue("EventSourceMinVersion", minVersion);
            command.Parameters.AddWithValue("EventSourceMaxVersion", maxVersion);
            connection.Open();

            using (SqlDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    var evnt = ReadEventFromDbReader(reader);
                    events.Add(evnt);
                }
            }
        }
    }

    return new CommittedEventStream(id, events);
}

internal static class Queries
```



```

        "Version", snapshot.Version
    );
    command.Parameters.AddWithValue(
        "Type", snapshot.GetType().AssemblyQualifiedName
    );
    command.Parameters.AddWithValue("Data", data);
    command.ExecuteNonQuery();
    }
}

transaction.Commit();
}
catch
{
    transaction.Rollback();
    throw;
}
}
}

internal static class Queries
{
    ...

    public const String InsertSnapshot =
        "DELETE FROM [Snapshots] " +
        "WHERE [EventSourceId] = @EventSourceId; " +
        "INSERT INTO [Snapshots] " +
        "    ([EventSourceId], [Timestamp], [Version], [Type], [Data])
        "VALUES (@EventSourceId, GETDATE(), @Version, @Type, @Data)";

}

```

Листинг 22.30. Логика загрузки моментального снимка в Ncqrs

```

public Snapshot GetSnapshot(Guid eventSourceId, long maxVersion)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        connection.Open();

        using (var command =
            new SqlCommand(Queries.SelectLatestSnapshot, connection))
        {
            command.Parameters.AddWithValue("@EventSourceId", eventSourceId);

            using (var reader = command.ExecuteReader())
            {
                if (reader.Read())
                {

```

```

        var snapshotData = (byte[])reader["Data"];

        using (var buffer = new MemoryStream(snapshotData))
        {
            var formatter = new BinaryFormatter();
            var payload = formatter.Deserialize(buffer);
            var theSnapshot = new Snapshot(
                eventSourceId, (long)reader["Version"], payload
            );

            return theSnapshot.Version > maxVersion
                ? null
                : theSnapshot;
        }
    }

    return null;
}
}
}

internal static class Queries
{
    ...

    public const String SelectLatestSnapshot =
        "SELECT TOP 1 * FROM [Snapshots] " +
        "WHERE [EventSourceId] = @EventSourceId " +
        "ORDER BY Version DESC";

    ...
}

```

Оправданно ли создание собственного хранилища событий?

На ранних этапах развития идеи регистрации событий не существовало никаких коммерческих инструментов. Разработчики были вынуждены конструировать собственные механизмы регистрации событий поверх существующих технологий, таких как реляционные или документоориентированные базы данных, как было описано ранее в этом разделе. Опыт показывает, что эта задача вполне решаема. Но с вводом более сложных сценариев, таких как проекции, сложные временные запросы и улучшенная масштабируемость, на их реализацию потребуется масса времени, которое можно было бы потратить на создание чего-то более ценного для предприятия. Именно поэтому у многих может появиться желание рассмотреть специализированные технологии, такие как Event Store Грега Янга, обеспечивающие множество дополнительных возможностей «из коробки».

Использование специализированного хранилища событий

Выбор существующего хранилища событий уменьшает объем работы, которую требуется выполнить, прежде чем приступить к реализации поддержки регистрации событий. Кроме того, выбрав готовое хранилище, такое как Event Store Грега Янга, вы получите множество дополнительных особенностей «из коробки», включая улучшенную поддержку проекций и многорежимной кластеризации для работы в высокомасштабируемых окружениях. Для демонстрации приемов использования хранилища событий примеры в этом разделе все так же будут ориентированы на службу предварительной оплаты, предлагаемую вымышленным оператором сотовой связи. Вы увидите, как сконструировать альтернативную реализацию `IEventStore` с применением клиентской библиотеки Event Store для C#, а также как выполнять запросы и получать проекции в веб-интерфейсе. Чтобы иметь возможность следовать за примерами в этом разделе, вам понадобится установить Event Store на свой компьютер.

Установка Event Store

Для использования проекций требуется версия Event Store не ниже 3.0.0 rc2. Загрузить Event Store можно по адресу <https://geteventstore.com/downloads/>. Загрузив zip-файл, распакуйте его в папку по своему выбору. Чтобы запустить Event Store, откройте консоль PowerShell с привилегиями администратора, перейдите в папку, куда был распакован архив, и выполните команду:

```
.\EventStore.SingleNode.exe --db .\ESData --run-projections=all
```

Для включения поддержки проекций потребуется выполнить несколько настроек. Откройте в браузере веб-интерфейс (<http://localhost:2113/projections>), перейдите на вкладку Projections (Проекции) и запустите следующие проекции: `$by_category` и `$stream_by_category` (для этого сначала щелкните на соответствующей ссылке, а затем — на кнопке Start (Пуск).)

ВНИМАНИЕ

Порядок установки Event Store практически в точности совпадает с шагами, описанными в главе 13 «Интеграция с RPC и REST посредством HTTP». Но имейте в виду, что версия 3.0.0 rc2, минимально необходимая для работы с проекциями, не готова для промышленного использования, хотя к моменту, когда вы будете читать эти строки, могут быть доступны более новые версии. Если вы будете использовать более новую версию, есть вероятность, что API проекций может измениться.

Использование клиентской библиотеки для C#

Чтобы воспользоваться преимуществами Event Store в своих приложениях, можно написать слой хранения данных, использующий официальную клиентскую библиотеку Event Store для C#. С другой стороны, Event Store поддерживает API,

действующий через гипертекстовый транспортный протокол (Hypertext Transport Protocol, HTTP), с которым вы познакомитесь в главе 26 «Запросы: предметная отчетность». В этом примере мы рассмотрим новую реализацию `IEventStore`, которая заменит версию для RavenDB, созданную ранее. Эта реализация опирается на следующие пакеты NuGet:

```
Install-Package EventStore.Client -version 2.0.2.0
Install-Package Newtonsoft.Json -version 6.0.3
```

`EventStore.Client` — это официальная клиентская библиотека, созданная разработчиками Event Store. Она взаимодействует с Event Store по протоколу TCP. `Newtonsoft.Json` — дополнительный пакет, необходимый для сохранения событий в формате JSON; то есть события будут преобразовываться из формата классов C# в формат JSON и обратно. В листинге 22.31 показан класс `GetEventStore` — конкретная реализация интерфейса `IEventStore` предоставляющая все необходимое.

Листинг 22.31. Реализация `IEventStore`, использующая в качестве хранилища Event Store

```
public class GetEventStore : IEventStore
{
    private IEventStoreConnection esConn;

    private const string EventClrTypeHeader = "EventClrTypeName";

    public GetEventStore(IEventStoreConnection esConn)
    {
        this.esConn = esConn;
    }

    public void CreateNewStream(string streamName,
        IEnumerable<DomainEvent> domainEvents)
    {
        // автоматически создает поток при добавлении события
        AppendEventsToStream(streamName, domainEvents, null);
    }

    public void AppendEventsToStream(string streamName,
        IEnumerable<DomainEvent> domainEvents, int? expectedVersion)
    {
        var commitId = Guid.NewGuid();

        var eventsInStorageFormat = domainEvents.Select(
            e => MapToEventStoreStorageFormat(e, commitId, e.Id)
        );

        esConn.AppendToStream(
            StreamName(streamName),
            expectedVersion ?? ExpectedVersion.Any,
            eventsInStorageFormat
        );
    }

    private EventData MapToEventStoreStorageFormat(object evtnt,
```

```

    Guid commitId, Guid eventId)
{
    var headers = new Dictionary<string, object>
    {
        // каждое событие в этой операции будет связано
        // с одним и тем же подтверждением
        {"CommitId", commitId},

        // сохранить тип класса, чтобы событие можно было восстановить
        // после загрузки из хранилища
        {EventClrTypeHeader, evnt.GetType().AssemblyQualifiedName}
    };

    // события и их заголовки хранятся в двоичном представлении JSON
    var data = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(evnt));
    var metadata = Encoding.UTF8.GetBytes(
        JsonConvert.SerializeObject(headers)
    );

    // если Event Store знает, что событие хранится в формате JSON,
    // в веб-интерфейсе предоставляется расширенная поддержка
    var isJson = true;

    return new EventData(eventId, evnt.GetType().Name, isJson, data, metadata);
}

public IEnumerable<DomainEvent> GetStream(string streamName,
    int fromVersion, int toVersion)
{
    // Event Store требует указать число извлекаемых событий,
    // а не наибольший номер версии
    var amount = (toVersion - fromVersion) + 1;
    var events = esConn.ReadStreamEventsForward(
        StreamName(streamName), fromVersion, amount, false
    );

    // преобразовать события обратно из JSON в DomainEvent.
    // Тип определяется заголовком
    return events.Events.Select(e => (DomainEvent)RebuildEvent(e));
}

private object RebuildEvent(ResolvedEvent eventStoreEvent)
{
    var metadata = eventStoreEvent.OriginalEvent.Metadata;
    var data = eventStoreEvent.OriginalEvent.Data;
    var typeOfDomainEvent =
        JObject.Parse(Encoding.UTF8.GetString(metadata))
            .Property(EventClrTypeHeader).Value;

    var rebuiltEvent = JsonConvert.DeserializeObject(
        Encoding.UTF8.GetString(data),
        Type.GetType((string)typeOfDomainEvent)
    );

    return rebuiltEvent;
}

```

```

}

// моментальные снимки в Event Store – это обычные события, хранящиеся
// в выделенных потоках; подробности см.:
// http://stackoverflow.com/questions/16359330/is-snapshotsupported-from-
// greg-young-eventstore
public void AddSnapshot<T>(string streamName, T snapshot)
{
    var stream = SnapshotStreamNameFor(streamName);
    var snapshotAsEvent = MapToEventStoreStorageFormat(
        snapshot, Guid.NewGuid(), Guid.NewGuid()
    );

    esConn.AppendToStream(stream, ExpectedVersion.Any, snapshotAsEvent);
}

public T GetLatestSnapshot<T>(string streamName) where T : class
{
    var stream = SnapshotStreamNameFor(streamName);
    var amountToFetch = 1; // только самый последний
    var ev = esConn.ReadStreamEventsBackward(
        stream, StreamPosition.End, amountToFetch, false
    );

    if (ev.Events.Any())
        return (T)RebuildEvent(ev.Events.Single());
    else
        return null;
}

private string SnapshotStreamNameFor(string streamName)
{
    // моментальные снимки – обычные события в отдельных потоках
    return StreamName(streamName) + "-snapshots";
}

private string StreamName(string streamName)
{
    // поддержка проекций в Event Store требует указывать единственный
    // дефис ("-"), подробности см. по адресу
    // https://groups.google.com/forum/#!msg/eventstore/D477bKLcdI8/62iFGHndMMI9
    var sp = streamName.Split(new []{ '-' }, 2);

    // удалить все дефисы, кроме первого
    return sp[0] + "-" + sp[1].Replace("-", "");
}
}

```

Потратим несколько минут, чтобы прояснить некоторые моменты в листинге 22.31. Обратите внимание, что при использовании готового хранилища событий потребовалось написать намного меньше программного кода, чем в примерах, использующих RavenDB или SQL Server, показанных выше в этой главе. Особенно показательным в этом отношении является извлечение событий и поддерж-

ка оптимистической формы управления одновременным доступом. Когда метод `GetStream()` извлекает события, он просто передает идентификатор потока, начальную версию и число событий. В решении на основе RavenDB для этого потребовалось сконструировать не самый простой запрос с несколькими предложениями. Такая простота объясняется тем, что Event Store изначально поддерживает понятие потока, а не эмулирует его на логическом уровне.

Наличие встроенной поддержки потоков также объясняет простоту реализации оптимистического управления одновременным доступом. Как вы могли заметить, достаточно просто передать в вызов `IEventStoreConnection.AppendToStream()` ожидаемый номер версии последнего хранящегося события, а о проверках и обо всем остальном позаботится само хранилище Event Store.

Единственное преимущество, которое действительно имеет RavenDB, — поддержка сериализации в формат JSON и обратно. В листинге 22.31 можно видеть, что `MapToEventStoreStorageFormat()` сохраняет в заголовке тип события — имя класса C#. Этот заголовок затем используется в `RebuildEvent()` при обратном преобразовании, когда события воссоздаются из формата JSON. В обоих случаях приходится вручную осуществлять сериализацию и десериализацию.

Поддержка моментальных снимков реализована в виде создания нового потока событий для каждого агрегата. Готовое решение Event Store также ярко проявляет себя в этой области. Как можно видеть в методе `GetLatestSnapshot()`, чтобы извлечь самый последний снимок, достаточно вызвать метод `IEventStoreConnection.ReadStreamEventsBackward()`, передав ему 1 в качестве числа возвращаемых событий.

Еще одна важная деталь, которая не была показана, — создание соединения с хранилищем Event Store. В листинге 22.32 продемонстрировано, как использовать для этого метод `EventStoreConnection.Create()` из клиентской библиотеки.

Листинг 22.32. Создание соединения с Event Store с помощью клиентской библиотеки

```
IPEndPoint endpoint = new IPEndPoint(IPAddress.Loopback, 1113);  
IEventStoreConnection con = EventStoreConnection.Create(endpoint)  
con.Connect();
```

Листинг 22.32 демонстрирует создание TCP-соединения с экземпляром Event Store, выполняющимся на локальном компьютере и принимающим соединения на порту 1113. Не забудьте изменить номер порта и адрес, указав значения, заданные в настройках Event Store.

ВНИМАНИЕ

Примеры в этой главе используют блокирующие методы из клиентской библиотеки Event Store для C#. Для достижения лучшей производительности и масштабируемости предпочтительнее применять неблокирующие, асинхронные методы с окончанием `Async` в именах (например: `AppendEventsToStreamAsync()`).

Временные запросы

Грег Янг и другие разработчики Event Store решили, что лучшим инструментом для выполнения запросов и создания проекций является JavaScript. Чуть ниже будут представлены примеры создания временных запросов в дружественном веб-интерфейсе Event Store. Но прежде нужно импортировать некоторые данные в хранилище. В загружаемых примерах к этой главе, в проекте `EventStoreDemo`, имеется класс `ImportTestData` с методом `TestMethod`, который можно выполнить из Visual Studio. Он запишет в хранилище Event Store несколько событий для коллекции агрегатов `PayAsYouGoAccount`. После выполнения этого теста вы сможете просматривать вновь созданные потоки и события в них на вкладке `Stream Browser` (Обозреватель потоков) в веб-интерфейсе (<http://localhost:2113/web/streams.htm>). Вы должны увидеть картину, изображенную на рис. 22.4, где показаны созданные потоки.

Recently Changed Streams

PayAsYouGoAccount-6f8ac515-9e3d-414f-b1f3-950e39256646

PayAsYouGoAccount-f727da8b-2a0c-4a8a-bae9-545285959ff5

PayAsYouGoAccount-7c296360-897b-43c2-b712-6b40041aa65b

PayAsYouGoAccount-21e6936b-afeb-4dd0-a9df-b695e8bf51d4

PayAsYouGoAccount-18722870-e0b9-493f-932f-5510e70f1721

Рис. 22.4. Содержимое вкладки `Stream Browser` (Обозреватель потоков) с успешно добавленными данными

Извлечение единственного потока

Несмотря на несколько необычное название, временные запросы не особенно сложны. Следующий короткий и очень полезный запрос сообщает клиенту, сколько минут он израсходовал в указанный день. Это может быть текущий день или некоторый день в прошлом. С его помощью клиент сможет оценить, насколько интенсивно он пользуется своим телефоном. В листинге 22.33 показан программный код на JavaScript, реализующий этот запрос.

Листинг 22.33. Запрос к хранилищу Event Store, возвращающий общее число минут, израсходованных клиентом в указанный день

```
fromStream('PayAsYouGoAccount-5b3415c8d58a4dcf955eed9978bcd8b1')
.when({
  // инициализировать состояние
  $init : function(s,e) {return {minutes : 0}},
```

```

"PhoneCallCharged" : function(s,e) {
    var dateOfCall = e.data.PhoneCall.StartTime;
    var june4th = "2014-06-04";
    if (dateOfCall.substring(0, 10) == june4th) {
        s.minutes += e.data.PhoneCall.Minutes.Number;
    }
});

```

Запросы в Event Store начинаются с определения запрашиваемых событий. Метод `fromStream()` в листинге 22.33 указывает, что запрос должен вернуть все события из одного потока, представляющего агрегат `PayAsYouGoAccount`. Идентификатор потока можно посмотреть на вкладке `Stream Browser` (Обозреватель потоков) в веб-интерфейсе. Каждое событие в потоке, начиная с самого раннего, затем передается в `when()`. Внутри `when()` с помощью шаблона определяется, какие события должны обрабатываться и как. В листинге 22.33 можно видеть, что обрабатываются только события типа `PhoneCallCharged`.

Результатом запроса к хранилищу Event Store является объект JavaScript, известный как состояние запроса (*state of the query*). Этот объект передается в `when()` с каждым событием, где обновляется в соответствии с логикой запроса. В листинге 22.33 видно, что состояние инициализируется в обработчике `$init` как объект JavaScript с единственным свойством `minutes`. Структура состояния для каждого вашего запроса определяется исключительно вами. Единственное требование — это должен быть допустимый объект JavaScript. Состояние используется в листинге 22.33 для суммирования в свойстве `minutes` значений времени во всех событиях `PhoneCallCharged`, представляющих телефонные звонки, выполненные 4 июня.

После выполнения запроса из листинга 22.33 во вкладке `Query` (Запрос) (<http://localhost:2113/web/query.htm>) можно увидеть результаты, изображенные на рис. 22.5.

На рис. 22.5 показано, что Event Store отображает результаты запроса в виде его заключительного состояния в панели **State** (Состояние) справа. Как уже упоминалось, структуру состояния определяете вы сами, и вы можете включить в него любые свойства, какие только потребуются. Например, запрос в листинге 22.33 можно расширить и подсчитать общее число израсходованных минут за несколько дней, где каждый день будет представлен отдельным свойством. Такой расширенный запрос приводится в листинге 22.34.

Листинг 22.34. Более сложный запрос, возвращающий состояние с множеством значений

```

fromStream('PayAsYouGoAccount-86e057b961624c6f92c9ab3c56e6e232')
.when({
    // инициализировать состояние
    $init : function(s,e) {return { june3rd: 0, june4th: 0, june5th: 0}},

    "PhoneCallCharged" : function(s,e) {
        var dateOfCall = e.data.PhoneCall.StartTime;
        var june3rd = "2014-06-03";
        var june4th = "2014-06-04";
        var june5th = "2014-06-05";

```

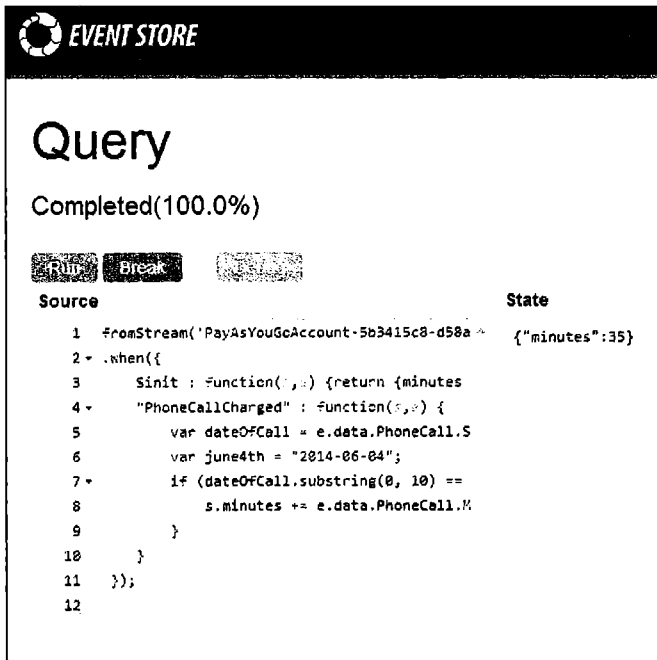


Рис. 22.5. Результаты выполнения запроса в веб-интерфейсе Event Store

```

if (dateOfCall.substring(0, 10) == june3rd) {
    s.june3rd += e.data.PhoneCall.Minutes.Number;
}
if (dateOfCall.substring(0, 10) == june4th) {
    s.june4th += e.data.PhoneCall.Minutes.Number;
}
if (dateOfCall.substring(0, 10) == june5th) {
    s.june5th += e.data.PhoneCall.Minutes.Number;
}
// обработать события других типов и обновить состояние соответственно
});

```

Состояние в листинге 22.34 содержит три свойства, соответствующие трем отдельным датам. Значение каждого из них увеличивается на число минут, использованных в телефонных звонках в соответствующие даты.

Выполнение запросов к нескольким потокам

Event Store позволяет выполнять запросы сразу к множеству потоков. Во многих сценариях использования бывает необходимо объединить данные из нескольких потоков, как это делает операция соединения таблиц в SQL. В листинге 22.35 показано, как объединить число минут за разные даты, чтобы найти общее число минут, израсходованных всеми клиентами в определенный день.

Листинг 22.35. Запрос, объединяющий события из множества потоков

```

fromCategory('PayAsYouGoAccount')
.when({
    // инициализировать состояние
    $init : function(s,e) {return { june3rd: 0, june4th: 0, june5th: 0}},

    "PhoneCallCharged" : function(s,e) {
        var dateOfCall = e.data.PhoneCall.StartTime;
        var june3rd = "2014-06-03";
        var june4th = "2014-06-04";
        var june5th = "2014-06-05";
        if (dateOfCall.substring(0, 10) == june3rd) {
            s.june3rd += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june4th) {
            s.june4th += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june5th) {
            s.june5th += e.data.PhoneCall.Minutes.Number;
        }
    }

    // обработать события других типов и обновить состояние соответственно
});

```

Метод `fromCategory()` в Event Store объединяет все потоки, идентификаторы которых начинаются с указанной строки, за которой следует дефис. Например, запрос в листинге 22.35 отберет все события во всех потоках, идентификаторы которых начинаются со строки `PayAsYouGoAccount-`. В необходимости наличия заключительного дефиса как раз и заключается один маленький недостаток Event Store. Именно по этой причине приватный метод `StreamName()` в классе `GetEventStore` (см. листинг 22.35) удаляет все дефисы, кроме первого.

Создание проекций

Вместо простого вычисления состояния иногда требуется получить подмножество событий и создать из них совершенно новый поток. Такая возможность особенно востребована в больших системах с миллионами или миллиардами событий. Возьмем для примера все того же оператора сотовой связи: если он будет обслуживать несколько миллионов клиентов, активно пользующихся телефонами, в хранилище легко может скопиться несколько миллиардов событий. Было бы весьма неэффективно применять запросы ко всему множеству событий. Проекция решает эту проблему, позволяя отобрать только события, представляющие интерес, поместить их в новый поток и затем применять запросы к вновь созданному потоку. В листинге 22.36 представлена проекция, группирующая все платежи, произведенные всеми клиентами, в новый поток `AllTopUps`.

Листинг 22.36. Проекция, выборочно копирующая события из множества потоков в один поток

```

fromCategory('PayAsYouGoAccount')
.when({

```



```

    "CreditAdded": function(s, event) {
        linkTo('AllTopUps', event);
    }
  });

```

После выполнения проекции из листинга 22.36 все события `CreditAdded`, присутствующие во всех потоках `PayAsYouGoAccount`*, будут добавлены в новый поток с идентификатором `AllTopUps`. Благодаря этому появляется возможность обрабатывать события `CreditAdded` без необходимости загружать все остальные события в потоках, такие как `PhoneCallCharged`. Ключевым инструментом поддержки такого поведения в Event Store является метод `linkTo()`, добавляющий в поток, имя которого соответствует первому аргументу (в данном примере `AllTopUps`), ссылку на указанное событие. Чтобы запустить проекцию, следует перейти на вкладку **Projections** (Проекции), выбрать **New Projection** (Создать проекцию) и заполнить форму, как показано на рис. 22.6.

The screenshot shows the 'New Projection' form in the Event Store web interface. The form is titled 'Collect All Top Ups'.

Name: Collect All Top Ups

Source:

```

1 fromCategory('PayAsYouGoAccount')
2 .when({
3   "CreditAdded": function(s, event) {
4     linkTo('AllTopUps', event);
5   }
6 });

```

Select Mode: One-Time

Checkpoints Enabled: ☐

Emit Enabled: ☒

Enabled: ☒

Post

Рис. 22.6. Создание проекции в веб-интерфейсе Event Store

На рис. 22.6 демонстрируется создание проекции с однократным запуском. Однако в поле **Select Mode** (Режим выбора) можно установить значение **Continuous** (Непрерывный). Это удобно, когда проекция должна автоматически включать также новые события, по мере их появления, почти в масштабе реального времени. Типичным примером, когда такая возможность может пригодиться, является реализация шаблона CQRS, о чем рассказывается чуть ниже.

Дополнительную информацию о возможностях проекций в Event Store можно найти в официальном блоге (<http://geteventstore.com/blog/>). Там имеется целая серия статей, посвященных проекциям (<http://geteventstore.com/blog/20130309/projections-8-internal-indexing/index.html>).

ПРИМЕЧАНИЕ

В главе 26 «Запросы: предметная отчетность» приводятся примеры создания отчетов с применением проекций Event Store.

Шаблон CQRS и регистрация событий

Для увеличения производительности и повышения масштабируемости кому-то из вас может понадобиться создать материализованные представления (*materialized views*) для своих событий. Такой подход избавляет от необходимости постоянно выполнять множество запросов для единственного потока событий. Вместо этого для каждого конкретного случая будут доступны предварительно вычисленные значения. Подобная проблема может возникать, например, когда имеется множество веб-страниц, каждая из которых выполняет свои запросы к единственному потоку событий, как показано на рис. 22.7.

С увеличением нагрузки на любую из веб-страниц будет увеличиваться нагрузка на единственный поток событий. А это означает, что менее важные страницы могут вызвать существенную деградацию производительности более важных страниц.

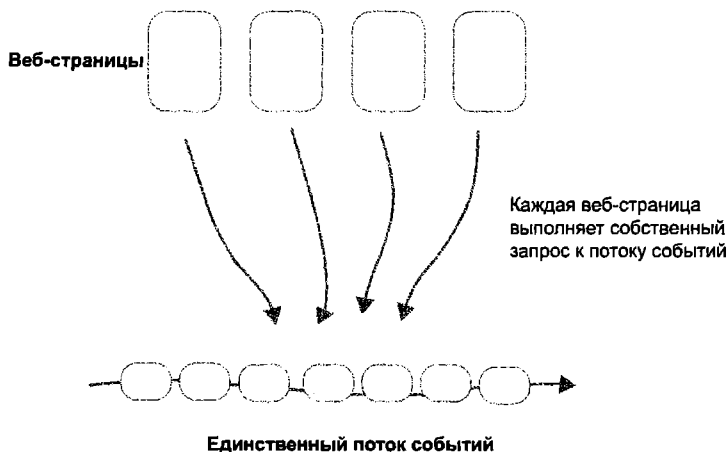


Рис. 22.7. Один поток событий используется для поддержки множества разных запросов и сценариев использования

Решение этой проблемы заключается в создании материализованного представления с данными для поддержки каждой страницы, как показано на рис. 22.8.

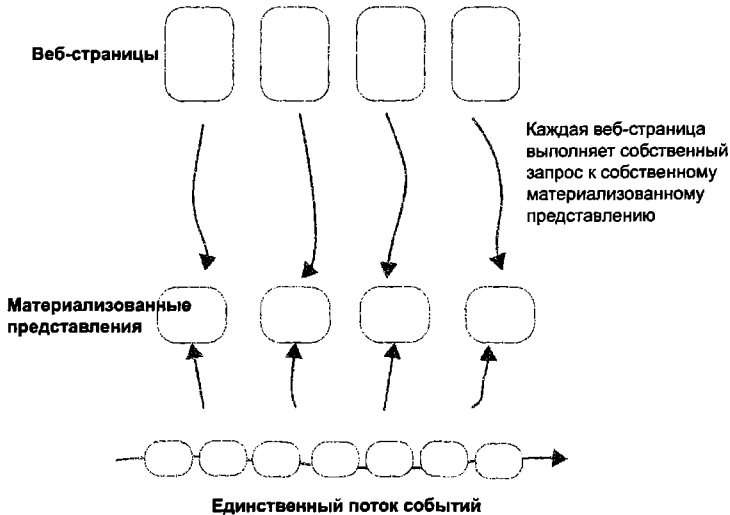


Рис. 22.8. Создание материализованных/ненормализованных представлений потока событий для каждого сценария использования

Решение на рис. 22.8 реализует шаблон CQRS (Command Query Responsibility Segregation — разделение ответственности команд и запросов), где команды (операции записи), проходя через предметную модель, помещаются в поток событий, а запросы (операции чтения) применяются к материализованным представлениям (также известным как кэшированные представления (view caches)). Это решение существенно смягчает проблему взаимовлияния разных сценариев использования, так как в каждом из них используется собственное кэшированное представление. А прямым его преимуществом является уменьшение конфликтов в борьбе за доступ к единственному потоку; теперь незапланированных обращений к потоку будет намного меньше (если таковые вообще останутся).

В этом разделе мы лишь мимоходом коснулись шаблона CQRS и его объединения с приемом регистрации событий. В главе 24 «CQRS: архитектура ограниченного контекста» будет представлено более полное обсуждение шаблона CQRS, включая возможность его применения к разным сценариям использования, не связанным с регистрацией событий.

Использование проекций для создания кэшированных представлений

Для реализации шаблона CQRS с приемом регистрации событий необходим некоторый способ создания ненормализованных представлений на основе потоков событий. Решение заключается в использовании проекций, представленных

выше. Материализованные представления на рис. 22.8 могут служить примером использования проекций совместно с шаблоном CQRS.

Как было показано ранее в этой главе, используя готовые инструменты, такие как Event Store, вы получаете всю необходимую функциональность для использования проекций и реализации шаблона CQRS. Однако, выбирая путь создания собственного хранилища событий, вам придется самостоятельно реализовать всю необходимую функциональность, что может оказаться очень непростой задачей. К счастью, RavenDB поддерживает возможность создания проекций (<http://ayende.com/blog/4530/raven-event-sourcing>).

Объединение CQRS с приемом регистрации событий

Несмотря на то что CQRS и регистрация событий являются самостоятельными понятиями и могут использоваться независимо друг от друга, убежденные последователи считают, что их объединение дает существенные преимущества. Рисунок 22.6 иллюстрирует, с какой простотой можно создавать кэшированные представления в виде проекций из потоков событий. Однако существуют и другие выгоды их совместного использования.

Потоки событий в виде очередей

В одном из примеров в главе 13 «Интеграция с RPC и REST посредством HTTP» было показано, что Event Store экспортирует потоки событий в виде лент Atom. В этом примере они служили альтернативой шине сообщений. Выгода такого подхода состоит в том, что хранилище событий избавляет от необходимости использовать очередь; точно так же оно избавляет от необходимости использовать любые другие технологии создания кэшированных представлений и организации ненормализованных данных.

Двухфазное подтверждение

С использованием Event Store в качестве основного источника данных, инструмента поддержки проекций и организации очередей сама собой отпадает необходимость в двухфазном подтверждении (или в распределенных транзакциях). Как только событие оказывается в потоке, оно благополучно помещается в очередь и сохраняется. Напротив, при организации хранилища событий в базе данных, их публикации в шине сообщений и обновлении кэшированного представления в другой базе данных, которые должны выполняться в рамках одной транзакции, вам придется позаботиться о тщательной обработке отказов в любой из операций и откатывать изменения, выполненные другими. Грег Янг подробно обсуждает эту проблему в своем блоге (<http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/>).

Еще раз о выгодах регистрации событий

Если вы впервые столкнулись с приемом регистрации событий, вам может потребоваться некоторое время на переваривание деталей и создание модели в своей

голове. Это совершенно понятно, и многие проходят через это. В этом разделе еще раз напоминаются выгоды, которые регистрация событий может принести в ваши проекты. Надеемся, что если вы и не запомните все детали, то у вас хотя бы сложится общее понимание — когда может пригодиться регистрация событий.

Конкурентные преимущества для предприятия

Эта глава началась с обсуждения фундаментального обоснования для использования регистрации событий: дополнительные конкурентные преимущества за счет возможности анализа не только текущих данных, но и всей хронологии событий, приведших к текущему состоянию. Если на рынке имеется две конкурирующие компании и одна из них использует прием регистрации событий для анализа, она сможет использовать имеющиеся знания для ускорения разработки своих продуктов и услуг.

Выразительность поведения агрегатов

Общение со специалистами в предметной области — важнейший аспект предметно-ориентированного проектирования и необходимое условие получения выгод от его применения. Для этого важно иметь предметную модель, которая выражает концептуальную модель с использованием единого языка. Как было показано ранее в этой главе, агрегаты с поддержкой регистрации событий выглядят практически как описательные инструкции, где каждый перегруженный метод `When()` читается как обычное предложение:

```
When {Предметное событие} {Применить бизнес-правила}
```

Это делает предметную модель еще более полезной в процессе переработки знаний. И вообще, такая реализация будет выглядеть намного более ясной для других разработчиков, плохо знакомых с предметной областью или проектом.

Простота хранения

Применение инструментов объектно-реляционного отображения (Object Relational Mappers, ORM) для сохранения агрегатов традиционно было источником противоречий и ошибок для многих коллективов разработчиков из-за несоответствий между предметными моделями и моделями данных. Как было показано в этой главе, хранение событий в потоках не сталкивается с этой проблемой, потому что никаких несоответствий нет. Это означает, что технология ORM не изолирует модель базы данных и не требует сложных преобразований. Фактически с применением регистрации событий появляется настоящая возможность менять технологии хранения, благодаря полиморфизму. Вы видели в этой главе, как можно переключаться между несколькими реализациями `IEventStore` по мере необходимости.

Следует отметить, что абстракция `IEventStore` имеет небольшой недостаток — она не гарантирует простоты переключения между технологиями хранения. Одним из примеров являются идентификаторы потоков; метод `fromCategory()` в `Event Store`

опирается на последний дефис в идентификаторах. Однако, как было показано, это ограничение легко обойти в `GetEventStore`.

Простота отладки

Ранее в этой главе вы видели, насколько проще в отладке системы, использующие регистрацию событий. Только задумайтесь — в вашем распоряжении вся хронология событий. Благодаря этому вы легко сможете восстановить любую их последовательность и найти событие, приведшее к неправильному изменению состояния или к ошибке какого-то другого типа. Вам не придется гадать, какая последовательность событий могла привести к проблеме.

Оценка ценности регистрации событий

Также не менее важно знать об отрицательных аспектах регистрации событий. Даже создатели хранилищ событий рекомендуют тщательно взвешивать все «за» и «против» перед их использованием. Проанализируйте, возможно, регистрация событий не окупится в вашем случае, особенно если ее применение влечет множество разнообразных проблем, решение которых потребует дополнительного времени.

Поддержка версий

По мере накопления знаний о предметной области и развитии продукта в предметную модель будут добавляться новые понятия и информация. Как результат, вам может понадобиться переименовать события, перераспределить данные между ними или внести какие-то другие изменения, влияющие на формат событий. Это представляет большую проблему, потому что к этому моменту у вас может накопиться значительный поток с событиями в старом формате. Данная проблема имеет решения, и они не всегда требуют больших усилий. Но поддержка разных версий может превратиться в настоящую проблему, если не уделять ей должного внимания.

Необходимость освоения новых понятий и обретения навыков

Проекции, временные запросы, моментальные снимки — многие понятия могут оказаться новыми для разработчиков, впервые начинающих использовать прием регистрации событий. Как обычно бывает в таких случаях, вам придется проверить теорию на практических экспериментах, прежде чем вы обретете опыт, поэтому будьте готовы к замедлению разработки в краткосрочной перспективе. Возможно, вам придется специально выделять время на изучение и эксперименты. Эти затраты также будут распространяться на всех, кто присоединяется к проекту и не имеет опыта использования регистрации событий.

Необходимость изучения новых технологий и овладения ими

Можно использовать для регистрации событий уже знакомые технологии хранения данных, такие как SQL Server, но весьма вероятно, что многие выберут специализированные хранилища. Изучение новых технологий также требует времени — потребуется не только их использование, но и развертывание на действующих серверах и мониторинг поведения и потребляемых ими ресурсов. Трудно дать какую-то количественную оценку, но обычно развертывание хранилища событий в первый раз на действующем сервере измеряется человеко-часами.

Более строгие требования к емкости хранилища данных

Очевидно, что для хранения всей хронологии событий требуется больше дискового пространства, чем для одного лишь текущего состояния. К счастью, память в наше время стоит невероятно дешево, поэтому весьма маловероятно, что для большинства это станет проблемой. Однако этот параметр нельзя упускать из виду и всегда следует пристально следить за ним.

Дополнительные ресурсы

- Обсуждение регистрации событий без привязки к какой-либо технологии на сайте проекта Event Store — <http://docs.geteventstore.com/>.
- Статья Мартина Фаулера о регистрации событий — <http://martinfowler.com/eaDev/EventSourcing.html>.
- Множество статей о регистрации событий в блоге Джереми Шассена (Jérémie Chassaing) — <http://thinkbeforecoding.com/tag/Event%20Sourcing>.
- Лев Городински (Lev Gorodinski) демонстрирует возможность применения DDD и регистрации событий на F# — <http://gorodinski.com/blog/2013/02/17/domain-driven-design-with-fsharp-and-eventstore/>.
- Регистрация событий с помощью библиотеки Akka (Scala или Java) — <http://doc.akka.io/docs/akka/snapshot/scala/persistence.html>.
- Группа DDD/CQRS в Google — <https://groups.google.com/forum/#!forum/dddcqrs>.

Ключевые идеи

- Регистрация событий заменяет традиционный способ хранения моментальных снимков состояния хранением полной хронологии событий, которые привели к текущему состоянию.
- Регистрация событий даст возможность воспроизвести состояние в любой момент в прошлом.

- Наличие хронологии дает мощную возможность анализа изменения состояния во времени — временные запросы.
- Временные запросы могут существенно увеличить потенциал предприятия за счет способности более глубокого проникновения в детали.
- Для применения регистрации событий предметные модели должны содержать событийно-ориентированные агрегаты.
- Агрегаты с поддержкой регистрации событий более наглядно отражают бизнес-правила и в меньшей степени зависят от выбора технологии хранения.
- Хранилище событий можно реализовать на основе уже используемых технологий, таких как документоориентированные базы данных и SQL Server.
- Event Store — специализированное хранилище событий, имеющее встроенную поддержку понятия потока и предоставляющее дополнительные функциональные возможности, такие как проекции.
- CQRS и регистрация событий образуют мощную комбинацию, в которой можно использовать проекции для создания кэшированных представлений.
- Освоение приема регистрации событий часто может не окупаться, поэтому не используйте его без внимательного изучения вопроса.

ЧАСТЬ IV

Шаблоны проектирования эффективных приложений

Глава 23: Конструирование пользовательских интерфейсов приложения

Глава 24: CQRS: архитектура ограниченного контекста

Глава 25: Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования

Глава 26: Запросы: предметная отчетность

23

Конструирование пользовательских интерфейсов приложения

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Введение в особенности создания пользовательских интерфейсов в слабосвязанных, распределенных и нераспределенных системах, создаваемых с применением приемов предметно-ориентированного проектирования
- Пример пользовательского интерфейса, извлекающего данные из множества ограниченных контекстов, которые действуют как единое приложение
- Пример пользовательского интерфейса, извлекающего данные из распределенных ограниченных контекстов

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 23 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Пользовательский интерфейс приложения — вот то, что обычно притягивает клиентов. Если он выглядит привлекательно и позволяет им достигать своих целей, например находить идеальное место для отдыха, они будут рады потратить свои деньги у вас. Но правильно построенный пользовательский интерфейс — это не только глянцевая картинка на экране. Есть также множество технических проблем, касающихся производительности, масштабируемости и слабой связанности ограниченных контекстов.

Одной из основных технических проблем пользовательских интерфейсов, является сбор всех данных воедино. В приложении электронной коммерции, например, может потребоваться показать на одной странице каталог товаров, цены, варианты доставки, специальные предложения и другие сведения. Как рассказывалось во второй части книги «Стратегические шаблоны: взаимодействия между ограниченными контекстами», в событийно-ориентированных приложениях информация хранится в множестве потенциально непротиворечивых ограниченных

контекстов. Там же говорилось, что такого рода системы не имеют разделяемых ресурсов; иными словами, веб-приложение не может просто выполнить запрос к базе данных другого ограниченного контекста, потому что это приводит к образованию тесной связи. Чтобы решить данную проблему, на выбор есть несколько вариантов, каждый из которых имеет свои достоинства и недостатки. Например, данные можно объединить на сервере или непосредственно в веб-странице, выполнив несколько AJAX-запросов. Ограниченные контексты могут возвращать данные в простом виде, обычно в формате XML или JSON, либо в виде разметки HTML, которую можно напрямую вставлять в веб-страницу. В этой главе будут показаны примеры каждого из упомянутых сценариев с рекомендациями, когда лучше использовать каждый из шаблонов, а также описания их достоинств и недостатков.

Однако прежде чем увидеть примеры, вы сначала познакомитесь с основными задачами, которые приходится решать при создании пользовательских интерфейсов, — от высокоуровневых решений, таких как выбор группы, которая должна отвечать за него, до низкоуровневых, таких как выбор языка программирования. К концу этой главы вы будете знать, как прикладной уровень обрабатывает входные данные, поступающие от пользовательского интерфейса, и как обеспечивает координацию взаимодействий с ограниченными контекстами.

Основы проектирования

Кого-то может удивить разнообразие подходов к проектированию пользовательских интерфейсов, извлекающих информацию из нескольких ограниченных контекстов. Некоторые из них могут влиять на организацию программных интерфейсов на сервере, а другие — даже определять выбор языка программирования. В действительности пользовательский интерфейс может даже определять, какие данные должны храниться в ограниченных контекстах.

Единые и составные пользовательские интерфейсы

Первое решение, которое нужно принять, приступая к проектированию пользовательского интерфейса, — определить, кто логически владеет им. Например, он может находиться внутри единственного ограниченного контекста (точнее, в виде единственного бизнес-компонента) и разрабатываться группой, отвечающей за этот ограниченный контекст. В другом случае представление может извлекать данные из множества ограниченных контекстов и не принадлежать ни одному из них.

Автономность

Пользовательский интерфейс для автономного приложения принадлежит единственному бизнес-компоненту. Ему не приходится извлекать данные из других ограниченных контекстов. Однако это означает, что бизнес-компоненту придется локально хранить всю информацию, представленную в пользовательском интерфейсе. Для этого ему необходимо подписаться на события, посылаемые другими

ограниченными контекстами, содержащими требуемые данные, и сохранять их у себя. Такая организация уже обсуждалась во второй части книги и показана на рис. 23.1.

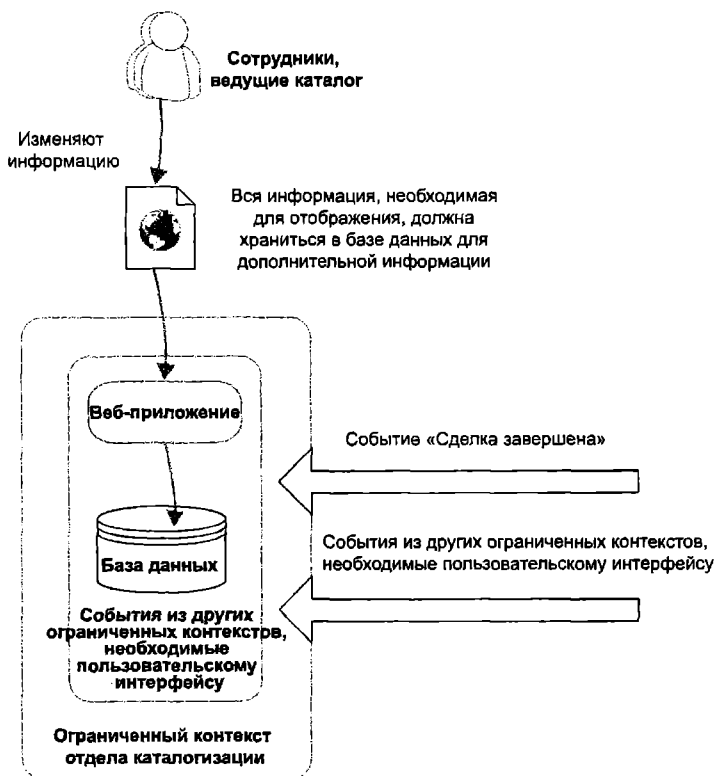


Рис. 23.1. Пользовательский интерфейс автономных приложений

На рис. 23.1 показано приложение, использующее дополнительные данные, которыми владеет ограниченный контекст отдела каталогизации. Такая организация позволяет работникам отдела изменять и дополнять информацию об определенных продуктах. Все сведения о продуктах хранятся в бизнес-компоненте, поэтому доступ к ним возможен всегда. Однако в процессе обновления информации сотрудники предприятия хотели бы знать объем продаж каждого продукта, чтобы понимать, имеет ли смысл прикладывать дополнительные усилия, направленные на улучшение качества информации. Эти сведения приобретаются путем подписки на событие «Сделка завершена», рассылаемой контекстом отдела продаж, и сохраняются в базе данных для последующего отображения в пользовательском интерфейсе автономного веб-приложения.

Потенциальная непротиворечивость может стать существенной проблемой для пользовательских интерфейсов в автономных приложениях, потому что отобра-

жаемая информация может не полностью соответствовать текущему моменту. Для сценария на рис. 23.1, например, нет ничего страшного, если информация об объемах продаж запоздает на минуты, часы или даже дни, потому что сотрудникам, ведущим каталог, достаточно лишь приближенного представления о популярности того или иного товара. Но если свежесть данных играет важную роль, лучшим выбором для организации пользовательского интерфейса может стать надежно согласованное приложение.

Авторитетность

Когда в едином пользовательском интерфейсе необходимо иметь самую свежую информацию из множества ограниченных контекстов, приложение должно напрямую обращаться к соответствующим ограниченным контекстам. Такую организацию можно видеть на рис. 23.2.

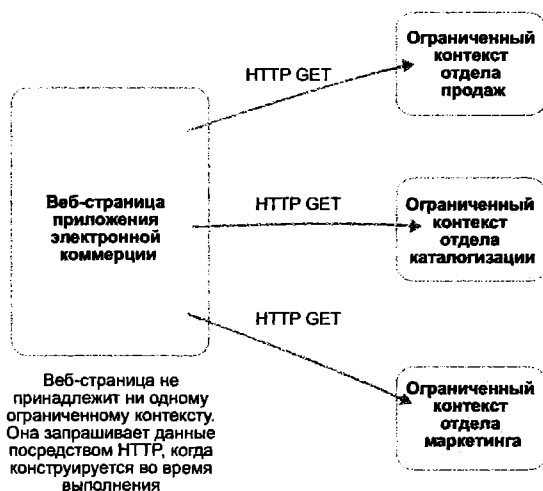


Рис. 23.2. Пользовательский интерфейс, опирающийся на авторитетность

Здесь изображена веб-страница приложения электронной коммерции, извлекающая специальные предложения, цены и другие сведения из разных ограниченных контекстов, каждый из которых является авторитетным источником своей доли информации. Это происходит всякий раз, когда очередной пользователь запрашивает страницу, поэтому она всегда содержит самые свежие сведения.

Пользовательские интерфейсы, извлекающие сведения по частям из нескольких авторитетных источников, не принадлежат какому-то одному ограниченному контексту. Для их реализации во многих компаниях создаются специальные группы разработчиков веб-интерфейсов, не вовлеченных в разработку ограниченных контекстов и отвечающих исключительно за веб-сайт. Если в вашем случае такой подход неприемлем, ответственность за создание пользовательских интерфейсов

можно возложить на разработчиков, занимающихся реализацией ограниченного контекста. При этом важно помнить, что концептуально пользовательский интерфейс не принадлежит их ограниченному контексту.

Некоторые рекомендации

Выбирая между автономностью интерфейса и авторитетностью отображаемой информации, не забывайте о взаимоотношениях между группами разработчиков. Если пользовательский интерфейс имеет прямое отношение к некоторому отделу, например занимающемуся созданием внутреннего инструмента, возможно, эффективнее будет оставить его разработку за этим отделом. С другой стороны, если пользовательский интерфейс является составной частью крупного приложения, имеющего множество пользовательских интерфейсов, таких как общедоступные веб-сайты, вам может потребоваться создать отдельную группу веб-разработчиков, которые будут заниматься вопросами веб-интерфейсов.

Необходимо также учитывать количество дополнительной информации и сложность ее сбора в автономном приложении. Если объем работ окажется слишком велик, а сценарии использования — относительно маловажными, вариант на основе авторитетных источников может оказаться предпочтительнее с точки зрения краткосрочных и долгосрочных усилий. И если в интерфейсе должна отображаться самая свежая информация, автономное приложение с его потенциальной непротиворечивостью будет, вероятно, не самым лучшим выбором.

HTML API и Data API

Конструируя веб-страницы из фрагментов HTML, возвращаемых каждым ограниченным контекстом, вы фактически перекладываете на них ответственность за то, как будут выглядеть и действовать соответствующие им области страницы, как показано на рис. 23.3.

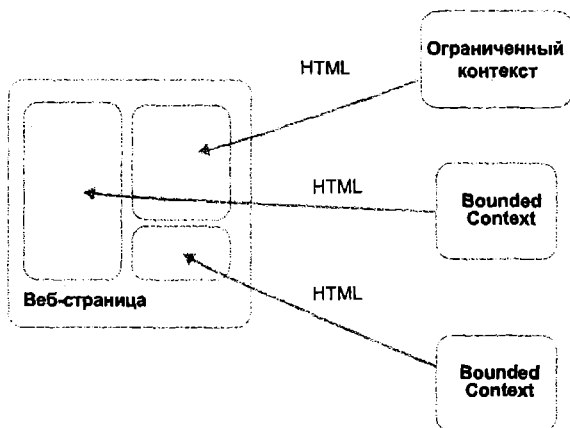


Рис. 23.3. Веб-страница, состоящая из фрагментов HTML, возвращаемых ограниченными контекстами

Другой вариант, уменьшающий влияние ограниченных контекстов на представление информации, — передача в страницы только данных. Этот альтернативный вариант позволяет сосредоточить решение всех задач представления в одном месте, как показано на рис. 23.4.

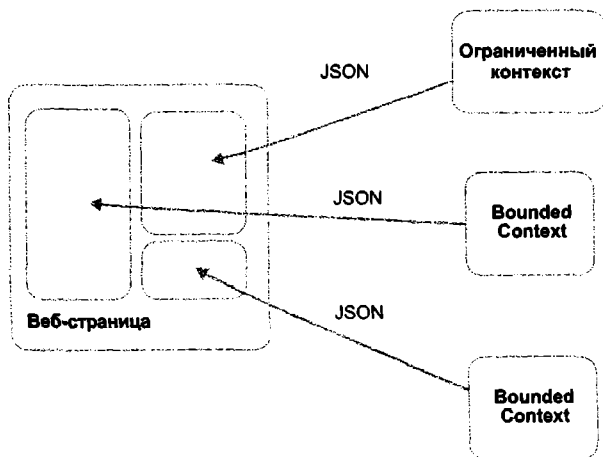


Рис. 23.4. Извлечение данных из нескольких ограниченных контекстов

Как показывает опыт, второй подход пользуется большей популярностью и обычно реализуется в виде JSON API. Тем не менее оба варианта вполне работоспособны. Один из важнейших вопросов, который следует учитывать, — будет ли экспортироваться программный интерфейс для внешнего использования. В главе 13 «Интеграция с RPC и REST посредством HTTP» рассказывалось о практике одновременного использования API для внутренних и внешних нужд («dogfooding») и ее преимуществах.

Координация/компоновка на стороне клиента и сервера

При создании пользовательского интерфейса, извлекающего информацию (в виде данных или фрагментов HTML) из нескольких ограниченных контекстов, компоновку этой информации можно выполнять и на стороне клиента, и на стороне сервера. С одной стороны, осуществляя множество AJAX-запросов внутри веб-страницы, можно избежать сложностей и появления дополнительных критических точек в серверном приложении. С другой стороны, это усложнит реализацию клиента на JavaScript. Один из вариантов ослабить данную проблему — создавать одностраничные приложения (Single Page Applications, SPA), как в настоящее время поступают многие коллективы. Обычно в таких ситуациях рекомендуют отдавать предпочтение реализации на стороне клиента, тем не менее оба подхода используются приблизительно одинаково широко. На рис. 23.5 демонстрируется компоновка информации на стороне клиента, а на рис. 23.6 — на стороне сервера.

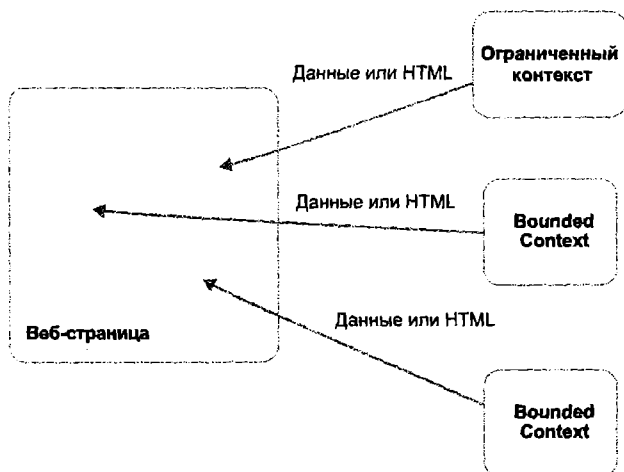


Рис. 23.5. Компоновка информации на стороне клиента

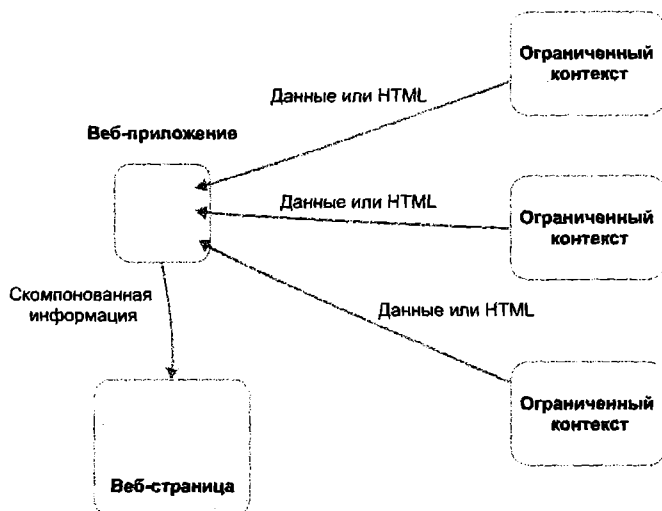


Рис. 23.6. Компоновка информации на стороне сервера

Пример 1: Пользовательский интерфейс на основе HTML API, с компоновкой информации из нераспределенных ограниченных контекстов на стороне сервера

Даже в том случае, когда все ограниченные контексты находятся в пределах одного решения и выполняются в рамках одного приложения, все еще может быть полезным разделить ответственность за оформление пользовательского интерфейса между ними. Когда одному ограниченному контексту потребуется изменить свою область страницы, он сможет сделать это, не мешая работе других ограниченных контекстов. Это в точности соответствует принципу единственной ответственности (SRP). В этом разделе будет представлена реализация данного сценария с применением `RenderAction()` из ASP.NET MVC в виде простой страницы, извлекающей фрагменты разметки HTML из трех ограниченных контекстов, входящих в состав одного решения, как показано на рис. 23.7.

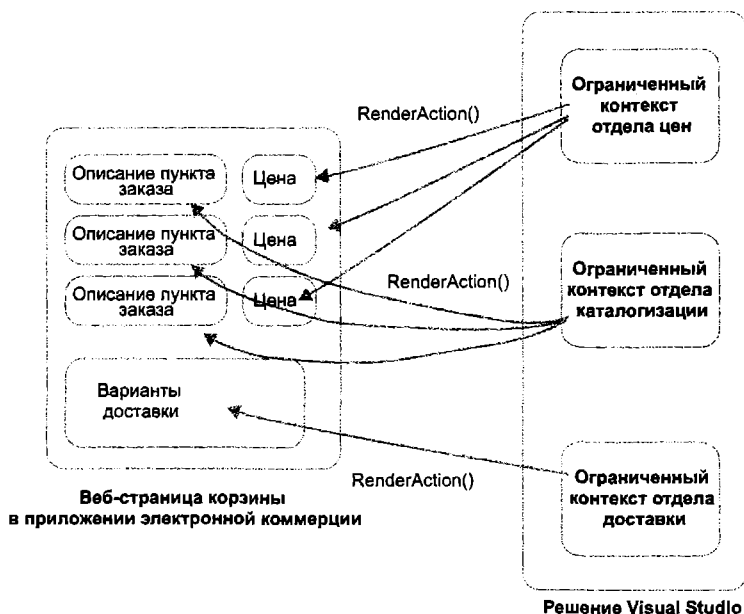


Рис. 23.7. Проект примера

Для начала создайте новое веб-приложение ASP.NET с именем `PPPPDDDD.NonDist.UIComp`. Выберите пустой шаблон и установите флажок MVC. Это приложение будет иметь только одно представление — составной пользовательский интерфейс, — поэтому для него вполне подойдет роль начальной страницы приложения. Для этого добавьте в папку `Controllers` класс `HomeController`, определение которого приводится в листинге 23.1.

ПРИМЕЧАНИЕ

Маршрут по умолчанию в ASP.NET MVC ищет метод `Index()` в контроллере с именем `HomeController`. Если в проекте имеется такой метод, он автоматически будет вызываться в ответ на обращение к базовому URL (/).

Листинг 23.1. Контроллер `HomeController`, возвращающий составной пользовательский интерфейс

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Контроллер `HomeController` просто возвращает представление, которое можно создать, добавив файл с именем `Index.cshtml` в папку `/Views/Home/` (ее следует создать вручную). После добавления представления замените содержимое файла разметкой из листинга 23.2.

Листинг 23.2. `/Views/Home/Index.cshtml` — составной пользовательский интерфейс

```
@{
    Layout = null;
    var productIdsInBasket = new string[3] { "prod1", "prod2", "prod3"};
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>PPPPDD Composite UI</title>
    @*
        Здесь также можно использовать RenderAction, если ограниченным
        контекстам понадобится внедрить свой код на javascript
    *@
</head>
<body>
    <div>
        <h1>Your Basket</h1>
        @foreach (var pid in productIdsInBasket)
        {
            <div class="basketItem" style="margin-bottom: 20px;">
```

```

        @{Html.RenderAction(
            "ItemInBasket", "catalogBoundedContext",
            new{productId=pid});}
        @{Html.RenderAction(
            "Price", "PricingBoundedContext",
            new{productid=pid});}
    </div>
    <br />
}
@{Html.RenderAction(
    "DeliveryOptions", "ShippingBoundedContext"); }
</div>
</body>
</html>

```

В листинге 23.2 показана простейшая реализация составного пользовательского интерфейса. Как можно заметить, сама страница является всего лишь шаблоном. Все информационное наполнение в виде фрагментов HTML она извлекает непосредственно из ограниченных контекстов с помощью `RenderAction()`. В первом аргументе ему передается имя метода в контроллере, название которого указывается во втором аргументе. То есть в данном примере первое обращение к `RenderAction()` приведет к вызову метода `CatalogBoundedContextController.ItemInBasket()`, которому будет передан идентификатор продукта в каталоге. Обратите внимание, что список идентификаторов продуктов жестко определен в начале страницы. Это помогает упростить пример и сконцентрировать его на аспектах, связанных с созданием пользовательского интерфейса, но на практике использовать такой прием не рекомендуется.

Чтобы обеспечить корректное отображение этой страницы, необходимо создать три контроллера с методами для вызова из `RenderAction()`. Сначала добавьте в папку `Controllers` класс `CatalogBoundedContextController`. Определение этого класса приводится в листинге 23.3.

Листинг 23.3. `CatalogBoundedContextController`

```

using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class CatalogBoundedContextController : Controller
    {
        [ChildActionOnly] // не может отображаться как самостоятельная страница
        public PartialViewResult ItemInBasket(string productId)
        {
            var product =
                SalesBoundedContext.ProductFinder.Find(productId);

            /* по соглашениям будет искать частичное представление:

```

```

        * /Views/CatalogBoundedContext/ItemInBasket.cshtml
        */
        return PartialView(product);
    }
}

// Следующий код в действительности можно было бы выделить в отдельный проект
namespace PPPDDD.NonDist.UIComp.SalesBoundedContext
{
    public static class ProductFinder
    {
        public static Product Find(string productId)
        {
            // имитировать поиск в базе данных
            return new Product
            {
                ID = productId,
                Name = "Product_" + productId,
                Description = "Lorem ipsum dolor sit amet",
                ImageUrl = "http://media.wiley.com/product_data/" +
                    "coverImage/84/04702927/0470292784.jpg"
            };
        }
    }

    public class Product
    {
        public string ID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public string ImageUrl { get; set; }
    }
}

```

Обратите самое пристальное внимание, что в листинге 23.3 этот контроллер извлекает всю необходимую информацию из ограниченного контекста отдела продаж вызовом методов объекта `ProductFinder`. Вообще говоря, весьма нежелательно вызывать из контроллера методы других ограниченных контекстов, потому что такой подход ведет к образованию тесных связей между ограниченными контекстами, из-за которых изменения в одном контексте могут отражаться на работе другого. Также это может привести к тому, что коллективы будут мешать друг другу, если они одновременно решат что-то изменить в реализациях своих контекстов.

Технически наиболее важной деталью является вызов `PartialView()`, который передает данные в модель представления. Он вернет разметку HTML из страницы фрагмента `/Views/CatalogBoundedContext/ItemInBasket.cshtml` (страница представления Razor). Теперь можно добавить в проект этот файл. Его содержимое приводится в листинге 23.4.

Листинг 23.4. ItemInBasket.cshtml

```
@model PPPDDD.NonDist.UIComp.SalesBoundedContext.Product

<div>
    <h3>@Model.Name</h3>
    <p>
        
        @Model.Description
    </p>
</div>
```

В листинге 23.4 определяется шаблон с использованием синтаксиса Razor, который заполняется фактическими значениями, полученными из модели представления. Разметка HTML, получаемая из этого шаблона, отображается непосредственно в составном пользовательском интерфейсе ItemInBasket в месте вызова `RenderAction()`.

Чтобы закончить этот пример, необходимо добавить в папку **Controllers** еще два контроллера: `PricingBoundedContextController` и `ShippingBoundedContextController`. Также требуется добавить частичные представления: `/Views/PricingBoundedContext.Price.cshtml` и `/Views/ShippingBoundedContext/DeliveryOptions.cshtml`. Содержимое всех этих файлов показано в листингах 23.5 — 23.8.

Листинг 23.5. PricingBoundedContextController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class PricingBoundedContextController : Controller
    {
        [ChildActionOnly] // не может отображаться как страница
        public PartialViewResult Price(string productId)
        {
            var price = PricingBoundedContext
                .PriceFinder
                .PriceFor(productId);

            /* по соглашениям будет искать частичное представление:
             * /Views/PricingBoundedContext/Price.cshtml
             */
            return PartialView(price);
        }
    }
}

// Следующий код в действительности можно было бы выделить в отдельный проект
namespace PPPDDD.NonDist.UIComp.PricingBoundedContext
```

```

{
    public static class PriceFinder
    {
        private static Lazy<Random> random = new Lazy<Random>();

        public static int PriceFor(string productId)
        {
            // имитировать поиск в базе данных
            return random.Value.Next(1, 1000);
        }
    }
}

```

Листинг 23.6. Price.cshtml

```

@model int

<div class="price">
    $@String.Format(Model.ToString(), "##.##")
</div>

```

Листинг 23.7. ShippingBoundedContextController

```

using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class ShippingBoundedContextController : Controller
    {
        [ChildActionOnly] // не может отображаться как страница
        public PartialViewResult DeliveryOptions()
        {
            var options = ShippingBoundedContext.DeliveryOptions.All();

            /* по соглашениям будет искать частичное представление:
             * /Views/ShippingBoundedContext/DeliveryOptions.cshtml
             */
            return PartialView(options);
        }
    }
}

// Следующий код в действительности можно было бы выделить в отдельный проект
namespace PPPDDD.NonDist.UIComp.ShippingBoundedContext
{
    using System.Collections.Generic;

    public static class DeliveryOptions
    {
        public static IEnumerable<DeliveryOption> All()
    }
}

```

```

{
    // имитировать поиск в базе данных
    return new List<DeliveryOption>
    {
        new DeliveryOption
        {
            ID = "ss1",
            Name = "Cheap & Cheerful",
            Price = 2,
            Duration = new Tuple<int,int>(7, 14)
        },
        new DeliveryOption
        {
            ID = "ss2",
            Name = "Super Fast",
            Price = 50,
            Duration = new Tuple<int,int>(1, 2)
        }
    };
}
}
}

public class DeliveryOption
{
    public string ID { get; set; }

    public string Name { get; set; }

    public int Price { get; set; }

    public Tuple<int, int> Duration { get; set; }
}
}

```

Листинг 23.8. DeliveryOptions.cshtml

```

@model IEnumerable<PPPPDD.NonDist.UIComp.ShippingBoundedContext
    .DeliveryOption>

<div class="deliveryOptions">
    <h2>Delivery Options</h2>
    @foreach(var option in Model)
    {
        <p>
            @Html.RadioButton("deliveryOptions", option.ID)
            @option.Name - @$@String.Format(
                option.Price.ToString(), "##.##")
            (@option.Duration.Item1 - @option.Duration.Item2 days)
        </p>
    }
</div>

```

Если вы запустите это приложение и откроете в браузере корневой URL, на вашем экране должен появиться составной пользовательский интерфейс, как показано на рис. 23.8.

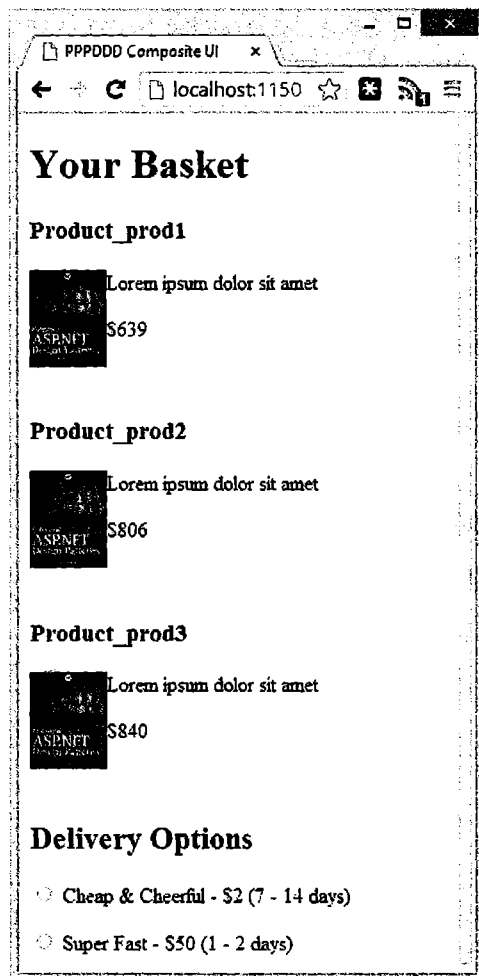


Рис. 23.8. Составной пользовательский интерфейс

Пример 2: Пользовательский интерфейс на основе DATA API, с компоновкой информации из распределенных ограниченных контекстов на стороне клиента

Создание веб-страниц, извлекающих информацию из разных HTTP API, — обычное дело в веб-разработке. В этом примере вы увидите, как получить информацию из разных ограниченных контекстов, каждый из которых выполняется в отдельном приложении, и объединить ее с использованием JavaScript непосредственно в браузере. Архитектура этого примера изображена на рис. 23.9.

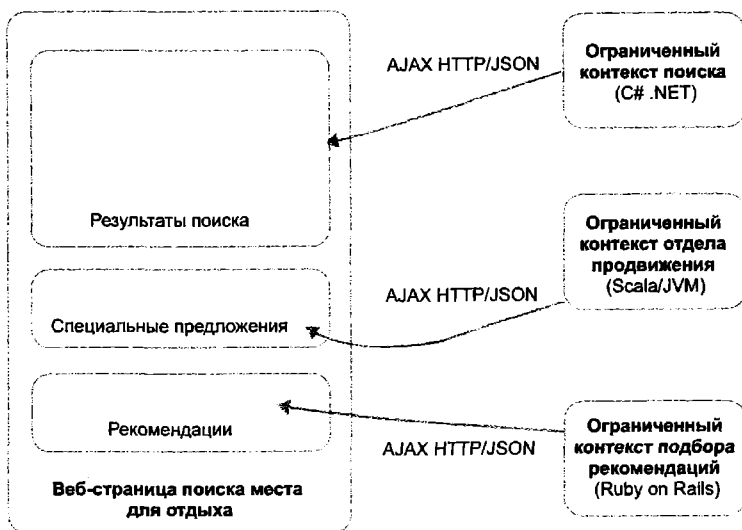


Рис. 23.9. Архитектура примера

Первое, на что следует обратить внимание на рис. 23.9: каждый API находится внутри ограниченного контекста, который ничего не знает о других ограниченных контекстах. Ограниченные контексты могут быть написаны с применением совершенно разных технологий, при условии, что реализуют необходимый HTTP API. Другая ключевая особенность в том, что каждый API возвращает данные в формате JSON. Это означает, что реализация представления данных ограничивается рамками главного веб-сайта.

ПРИМЕЧАНИЕ

В процессе изучения этого примера вы заметите, что все API находятся внутри одного и того же проекта ASP.NET и фактически не являются отдельными приложениями, как показано на рис. 23.9. Это помогло сделать пример сосредоточенным на особенностях создания пользовательского интерфейса, но помните, что этот пример имитирует архитектуру, изображенную на рис. 23.9, где каждый API реализован в отдельном приложении.

Для начала создайте новое веб-приложение ASP.NET с именем PPPDDD Dist. UIComp. Как и прежде, создавая проект, выберите пустой шаблон и установите флажок MVC. Создав проект, добавьте контроллер HomeController и соответствующее ему представление /Views/Home/Index.cshtml, которое автоматически будет выбрано в качестве страницы по умолчанию для приложения. Определение HomeController приводится в листинге 23.9, а содержимое Index.cshtml — в листинге 23.10.

Листинг 23.9. HomeController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Листинг 23.10. /Views/Home/Index.cshtml

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>PPPPDD Distributed Composite UI</title>
    <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
    <script src="/Scripts/pppddd-application.js" type="text/javascript">
    </script>
</head>
<body>
    <div>
        <p>You searched for: Moderately Priced Autumn Sun</p>
        <div id="holidays">
            <h3>Holidays Matching Your Search</h3>
        </div>
        <div id="promotions">
            <h3>Special Offers</h3>
        </div>
        <div id="recommendations">
            <h3>Holiday Recommendations Just For You</h3>
        </div>
    </div>
</body>
</html>
```

```
        </div>
    </div>
</body>
</html>
```

Контроллер `HomeController` из листинга 23.9 отображает представление, разметка которого приводится в листинге 23.10. Обратите внимание, что здесь представление имеет больше содержимого, чем в примере 1. Это объясняется тем, что здесь ограниченные контексты поставляют только данные, а не готовую разметку HTML. Часть разметки добавляется динамически по мере извлечения данных из ограниченных контекстов. Это происходит внутри сценария `pppddd-application.js` на JavaScript, ссылка на который присутствует в заголовке страницы. Содержимое `pppddd-application.js` показано в листинге 23.11. Сохраните этот файл в папке `Scripts`.

Листинг 23.11. /Scripts/pppddd-application.js

```
function createHolidayView(holiday) {
    return '<div>' +
        '' +
        '<h4>' + holiday.Title + '</h4>' +
        '$' + holiday.Price + 'pp' +
        '</div>' +
        '<br />';
}

// когда страница загрузится
$(document).ready(function() {

    // запросить места проведения отдыха, симитировав поиск
    $.getJSON("/holidays", function (json) {
        $.each(json, function (index, holiday) {
            $('#holidays').append(createHolidayView(holiday));
        });
    });

    // запросить специальные предложения
    $.getJSON("/promotions", function (json) {
        $.each(json, function (index, holiday) {
            $('#promotions').append(createHolidayView(holiday));
        });
    });

    // запросить рекомендации для данного пользователя
    $.getJSON("/recommendations", function (json) {
        $.each(json, function (index, holiday) {
            $('#recommendations').append(createHolidayView(holiday));
        });
    });
});
```

```
/*  
 * JQuery используется здесь исключительно в демонстрационных целях.  
 * Другие фреймворки, такие как Angular.js, Knockout.js и пр., могут  
 * оказаться более удачным выбором.  
 */
```

После загрузки страницы сценарий `pppddd-application.js` выполняет веб-запрос к каждому ограниченному контексту. Получив очередной ответ в формате JSON, он обновляет соответствующую часть страницы. Одна важная деталь — функция `createHolidayView()`. Обратите внимание, что она генерирует разметку HTML для отображения в странице. Благодаря этому все задачи, связанные с представлением информации, решаются исключительно внутри веб-приложения, а не разбрасываются по разным ограниченным контекстам, как это обычно происходит для API, возвращающих HTML. (Это не лучше и не хуже; это осознанный выбор, который вам придется сделать, взвесив все «за» и «против».)

ВНИМАНИЕ

Листинг 23.11 немного упрощен в целях уменьшения сложности примера. Одно из таких упрощений предполагает, что каждый HTTP API возвращает данные в одном и том же формате JSON (показан в листинге 23.12). В действующих системах такое маловероятно и к тому же не приветствуется из-за образования тесной связи. Другое упрощение — создание разметки HTML в функции `createHolidayView()`. В действующих приложениях может быть предпочтительно использовать библиотеки поддержки шаблонов, такие как `handlebars.js` (<http://handlebarsjs.com/>).

Листинг 23.12. Формат JSON для результатов поиска, используемый в этом примере

```
[  
  {  
    "Title": "...",  
    "Price": xx,  
    "ImgUrl": "http://..."  
  },  
  ...  
]
```

Пользовательский интерфейс готов. Единственное, чего еще не хватает, это реализации HTTP API, поддерживаемого каждым ограниченным контекстом. В настоящей распределенной системе каждый такой API действовал бы в отдельном приложении внутри ограниченного контекста. Но в этом примере для простоты все они находятся внутри одного приложения. Чтобы включить эти API в проект, добавьте в папку `Controllers` классы `HolidaysController`, `PromotionsController` и `RecommendationsController`. Определения этих классов показаны в листингах 23.13–23.15.

Листинг 23.13. HolidaysController

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
     * Этот контроллер представляет API ограниченного контекста поиска.
     * Он мог бы выполняться как отдельное приложение и определяться в
     * другом проекте, отдельно от других API, в данный момент
     * включенных в этот проект
     */
    public class HolidaysController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "2 Weeks in Rhodes",
                    Price = 688,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "1 Week in Barbados",
                    Price = 320,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
        {
            public string Title { get; set; }
            public int Price { get; set; }
            public string ImgUrl { get; set; }
        }
    }
}
```

Листинг 23.14. PromotionsController

```

using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
     * Этот контроллер представляет API ограниченного контекста
     * продвижения. Он мог бы выполняться как отдельное приложение
     * и определяться в другом проекте, отдельно от других API,
     * в данный момент включенных в этот проект
     */
    public class PromotionsController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "Relaxing Med Cruise",
                    Price = 999,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "Romantic Weekend Break in Paris",
                    Price = 120,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
        {
            public string Title { get; set; }
            public int Price { get; set; }
            public string ImgUrl { get; set; }
        }
    }
}

```

Листинг 23.15. RecommendationsController

```

using System;
using System.Collections.Generic;

```

```
using System.Web.Mvc;
namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
     * Этот контроллер представляет API ограниченного контекста
     * рекомендаций. Он мог бы выполняться как отдельное приложение
     * и определяться в другом проекте, отдельно от других API,
     * в данный момент включенных в этот проект
     */
    public class RecommendationsController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "2 Weeks in Mykonos",
                    Price = 450,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "2 Weeks in Kos",
                    Price = 365,
                    ImgUrl = "http://media.wiley.com/product_data/" +
                        "coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
        {
            public string Title { get; set; }
            public int Price { get; set; }
            public string ImgUrl { get; set; }
        }
    }
}
```

Как можно увидеть, каждый контроллер реализует лишь необходимый минимум, чтобы вернуть результат в требуемом формате (см. листинг 23.12). В действующих приложениях здесь обычно вызываются прикладные службы. Как вы узнаете в следующей главе, прикладные службы находятся между предметной моделью и внешними контрактами, такими как HTTP API, для координации действий на основе данных, поступающих из пользовательского интерфейса или передаваемых в пользовательский интерфейс. Прежде чем перейти к следующей главе,

нажмите клавишу F5 в своем проекте и убедитесь, что страница отображается, как показано на рис. 23.10.



Рис. 23.10. Компоновка JSON API на стороне клиента

Ключевые идеи

- Организация ограниченных контекстов может оказывать существенное влияние на пользовательский интерфейс, и наоборот.
- Выбор группы, которая должна отвечать за реализацию пользовательского интерфейса, может значительно повлиять на работу ее членов и принимаемые ими технические решения.
- Извлечение данных из множества ограниченных контекстов может выполняться и на стороне клиентов, в сценариях на JavaScript, и на стороне сервера, с использованием любой предпочтительной технологии.

- Компоновка на стороне клиента помогает уменьшить сложность серверного компонента и ослабить зависимость от него.
- Компоновка на стороне сервера избавляет от необходимости использовать JavaScript и устраняет ограничения производительности браузеров.
- Пользовательские интерфейсы можно конструировать из фрагментов HTML или на основе данных, извлекаемых из ограниченных контекстов, обычно в формате JSON или XML.
- Компоновка из фрагментов HTML дает больший контроль ограниченными контекстами, однако дробит задачи представления.
- Компоновка из данных изолирует задачи представления рамками единого веб-приложения, однако лишает ограниченные контексты возможности влиять на их отображение.

24

CQRS: архитектура ограниченного контекста

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Проблемы использования единой модели для реализации сложного представления и предметной логики
- Как шаблон CQRS разделяет модели чтения и записи
- Распространенные ошибочные представления о шаблоне CQRS
- Как шаблон CQRS помогает масштабировать ограниченные контексты

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 24 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

CQRS (Command Query Responsibility Segregation — разделение ответственности команд и запросов) — это простой шаблон, который можно применить к ограниченному контексту. Он делит предметную модель на две модели: модель чтения и модель записи (иногда ее называют транзакционной моделью).

Цель такого разделения — позволить модели обслуживать все потребности одного ограниченного контекста без компромиссов. Два контекста, которые получают в результате, сообщают о состоянии предметной области и выполняют деловые операции, также их называют сторонами чтения и записи (read and write sides). Использование единой модели для ограниченных контекстов, требующих сложного представления и имеющих богатую предметную логику, часто приводит к получению чрезмерно сложной модели, лишенной целостности, вызывающей путаницу в умах специалистов в предметной области и лежащейся непосильным

грузом сопровождения на плечи разработчиков. Применив шаблон CQRS, модель можно разбить на две части и оптимизировать каждую часть в отдельности, чтобы обеспечить более эффективную работу каждого контекста.

CQRS не является высокоуровневой архитектурой; это шаблон для уменьшения сложности, который можно применять к ограниченным контекстам для поддержки модели представления, не соответствующей транзакционной модели. В большинстве веб-приложений наблюдается несовпадение между запросами и командами. Шаблон CQRS разбивает такие модели на две и позволяет оптимизировать каждую из сторон независимо друг от друга.

В этой главе вы познакомитесь с шаблоном CQRS и получите общее представление о том, в каких ситуациях его применение может быть эффективно. Более подробно о реализации сторон команд и запросов рассказывается в главе 25 «Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования» и главе 26 «Запросы: предметная отчетность».

Проблемы использования единой модели для двух контекстов

На рис. 24.1 изображена типичная многослойная архитектура ограниченного контекста. В основе ее лежит предметная модель. Предметная модель создается для поддержания предметных инвариантов при обработке транзакций. Она состоит из нескольких агрегатов предметных объектов, созданных для обеспечения непротиворечивости и выражения правил и логики предметной области. Для нужд представления, однако, структура агрегатов может оказаться непригодной из-за необходимости загружать множество разных агрегатов в прикладных службах для наполнения моделей представления, которые могут содержать только подмножество данных, извлекаемых из экземпляров агрегатов. Создание представлений в таких случаях может быстро превратиться в сложную и запутанную задачу и порой даже замедлить работу системы.

Для поддержки представления информации предметные модели вынуждены экспортировать внутреннее состояние и включать свойства представления, которые не имеют ничего общего с инвариантами предметной области. Репозитории часто содержат множество дополнительных методов для нужд представления, например, для постраничного извлечения данных, выполнения запросов и поиска по произвольному тексту. Поскольку сторона чтения приложения обычно используется чаще, чем сторона записи, увеличение производительности отчетов оказывается более востребованным для пользователей. В попытках упростить и ускорить сторону чтения разработчики часто отказываются от предметной модели, объединяя агрегаты и используя отложенную загрузку, чтобы предотвратить извлечение данных, ненужных для выполнения деловых операций в транзакциях, но необходимых для представления. В результате такого подхода получается единая модель, полная компромиссов и плохо подходящая для обеих сторон, как чтения, так и записи.

Архитектура, более подходящая для сложных ограниченных контекстов

На рис. 24.2 изображена архитектура, использующая шаблон CQRS. Она удовлетворяет противоречивые потребности двух контекстов, а именно контекстов чтения и записи, предоставляя две отдельные модели вместо одной. Каждая модель теперь может оптимизироваться под конкретный контекст и при этом оставаться концептуально целостной. В некотором смысле происходит применение шаблона ограниченного контекста на более низком уровне, когда одна модель создается для контекста чтения, а другая модель — для контекста записи. На рис. 24.2 видно разделение между командами и запросами; между ответствен-

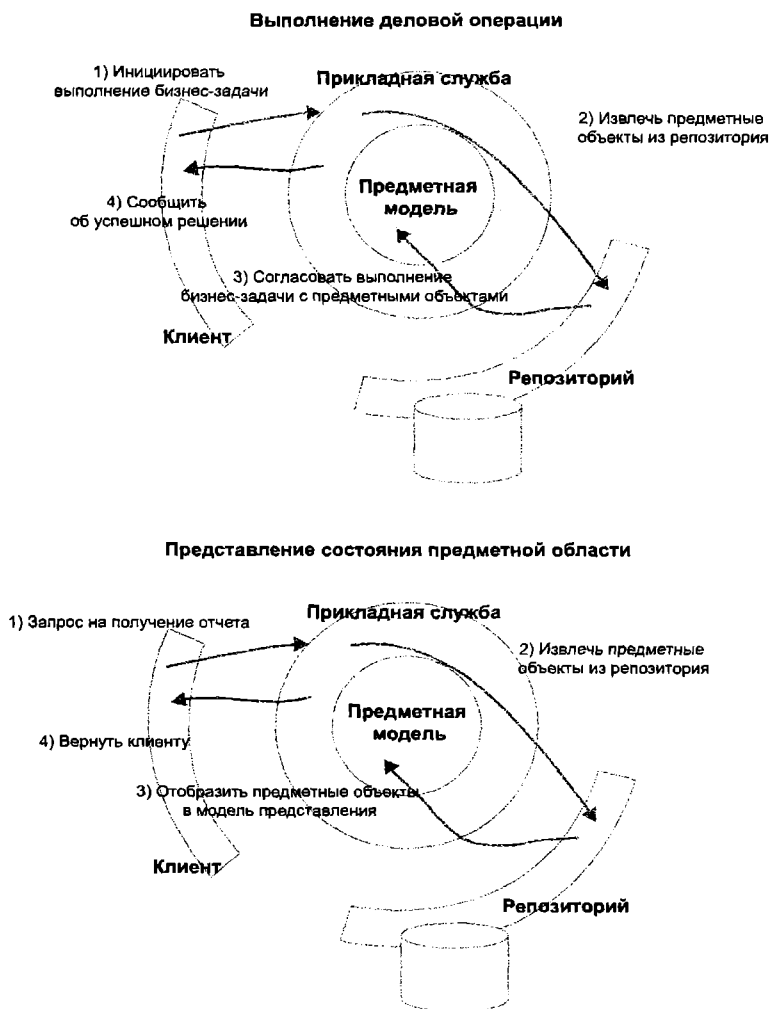


Рис. 24.1. Единственная модель включает стороны чтения и записи приложения

ностью за выполнение бизнес-задач, вызываемых клиентом, и ответственностью за выполнение запросов для получения отчетов. Стороны чтения и записи на рис. 24.2 используют одно и то же хранилище данных. Однако это совершенно не обязательно; для лучшей масштабируемости сторона чтения может использовать свое, отдельное хранилище.

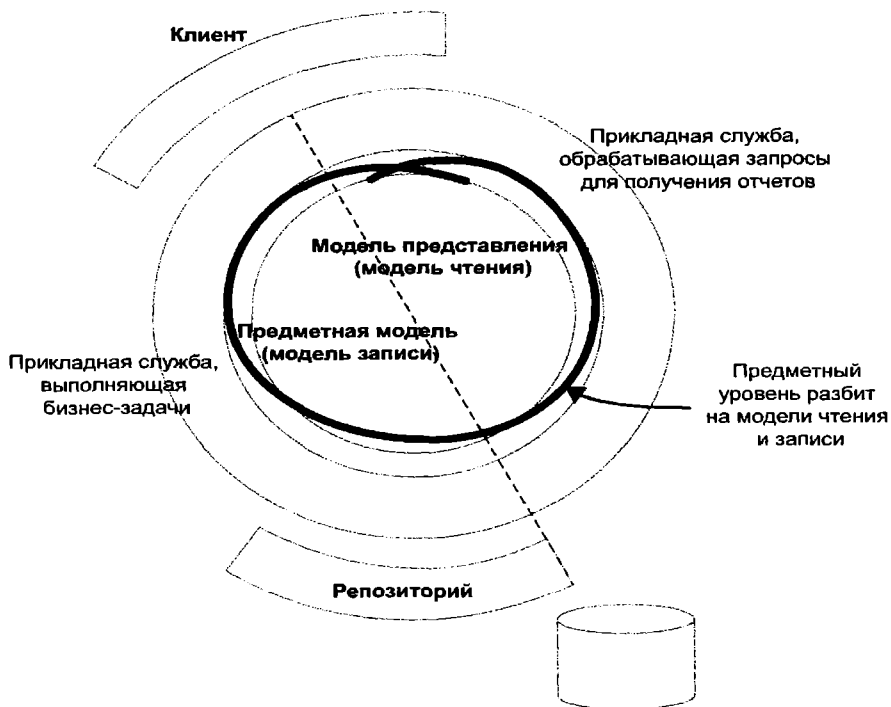


Рис. 24.2. Шаблон CQRS с разными моделями для чтения и записи

Команды: бизнес-задачи

Задачей стороны команд в архитектуре является поддержка правил предметной области. Она представляет предметную логику, соответствующую бизнес-задачам. Архитектура, изображенная на рис. 24.3, на первый взгляд выглядит как типичная многослойная архитектура; однако сторона команд не поддерживает запросы, а любые ответы на команды всего лишь подтверждают успешное выполнение бизнес-задач, инициированных клиентом.

Явное моделирование намерений

Команда представляет бизнес-задачу, сценарий использования системы и находится на уровне прикладных служб. Команды следует писать на языке, понятном специалистам предприятия. Это не единый язык; это язык, отражающий пове-



Рис. 24.3. Сторона команд в CQRS

дение системы, а не термины и понятия предметной модели. Обычно при использовании методики разработки BDD команды вытекают из сценариев использования и историй, которые вы производите.

Команды должны моделироваться как глаголы, а не как существительные. Они должны явно выражать намерение пользователя. Пример команды показан в листинге 24.1. Команда — это простой объект переноса данных (Data Transfer Object, DTO) с простой проверкой параметров.

Листинг 24.1. Пример объекта команды

```

public class CustomerWantsToRedeemAGiftCertificate
{
    public CustomerWantsToRedeemAGiftCertificate(Guid accountId,
                                                string giftCertificate)
    {
        AccountId = accountId;
        GiftCertificate = giftCertificate;
    }
    public Guid AccountId { get; private set; }
    public string GiftCertificate { get; private set; }
}

```

Как можно видеть в листинге 24.1, название команды явно отражает намерения пользователя. В данном случае — желание погасить подарочный сертификат. Команда представляет запрос на выполнение деловой операции, поэтому записывается в настоящем времени (например: «я хочу сделать что-либо»), в противоположность предметным событиям, выражающим действие в прошлом времени (например: «что-то произошло»).

Модель, свободная от задач представления

Модель, решающая задачи, связанные с представлением и транзакциями, часто несет на себе отпечаток пользовательского интерфейса приложения. Агрегаты, образованные для поддержки инвариантов, меняют свою форму, превращаясь в структуры, соответствующие потребностям пользовательского интерфейса. Например, взгляните на модель пользовательского интерфейса, изображенную на рис. 24.4. Эта типичная страница с суммарной информацией отображает различные атрибуты клиента.

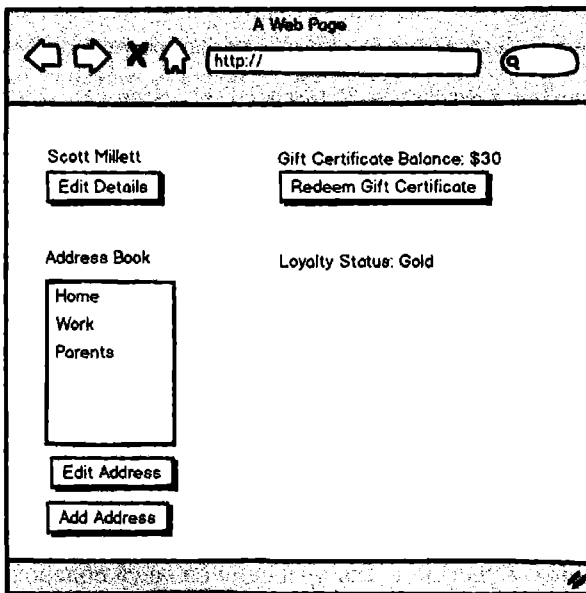


Рис. 24.4. Пользовательский интерфейс извлекает данные из множества агрегатов

В листинге 24.2 приводится определение типа предметного объекта, который используется для представления и одновременно реализует предметную логику.

Листинг 24.2. Предметный объект, используемый в реализации поведения и в пользовательском интерфейсе

```
public class Customer
{
```

```
// ...
public ContactDetails ContactDetails { get; private set; }
public LoyaltyStatus LoyaltyStatus { get; private set; }
public Money GiftCertBalance { get; private set; }
public IEnumerable<Address> AddressBook { get; private set; }
}
```

Теперь сконструировать модель представления очень просто, потому что предметная модель полностью соответствует потребностям пользовательского интерфейса. В листинге 24.3 демонстрируется простой случай извлечения полного агрегата, представляющего клиента и его отображение, в модель представления.

Листинг 24.3. Модель представления, необходимая только для пользовательского интерфейса

```
public class CustomerDashBoardView
{
    // ...
    public CustomerViewModel Generate(Guid customerId)
    {
        var customer = _customerRepository.FindBy(customerId);

        var customerView = MapCustomerViewModelFrom(customer);

        return customerView;
    }
}
```

Однако агрегат получился слишком большим — он хранит всю информацию, имеющую отношение к клиенту. Этот агрегат сконструирован на основе потребностей пользовательского интерфейса, а не инвариантов предметной области. Иными словами, он определен на основе экранной формы, а не предметной функциональности. Этих проблем можно избежать, применив шаблон CQRS и избавив модель от любых требований, связанных с представлением. Всеобъемлющее понятие клиента не дает никаких преимуществ модели команд, его реализация выглядит неуклюжей, и такие реализации обычно называют «божественными объектами» (god object). Гораздо предпочтительнее иметь один агрегат, ответственный за реализацию правил определения лояльности, второй агрегат — с дополнительными сведениями о клиенте и третий — для хранения баланса подарочных сертификатов. Кроме того, единый язык лучше организовывать на основе поведения приложения, а не на его пользовательском интерфейсе. Специалисты в предметной области общаются в терминах логики и правил предметной области, а не пользовательского интерфейса.

Модель команд без дополнительных обязанностей обычно получается в большей степени сосредоточенной на поведении прикладных служб, а агрегаты — более компактными. Репозитории могут быть существенно упрощены, так как ограничиваются извлечением агрегатов по их идентификаторам и не должны предоставлять разнообразные методы запросов, такие как постраничная выборка и сортировка. Разработчики могут моделировать агрегаты на основе инвариантов и все

внимание уделить поведению транзакций, не отвлекаясь на потребности пользовательского интерфейса.

Обслуживание бизнес-запросов

Обработчик команд — это разновидность прикладных служб. Службы обрабатывают команды и содержат логику, управляющую решением задачи. Эта логика может делегировать выполнение операций предметной модели, сохранять и извлекать предметные объекты и вызывать инфраструктурные службы, такие как пересылка клиентам по электронной почте результатов взаимодействий с платежной системой.

Обработчики команд могут возвращать только подтверждение об успехе или неудаче выполнения команды; они не должны использоваться для получения информации о состоянии предметной модели. В листинге 24.4 приводится пример обработчика команд. Подробнее о реализации команд рассказывается в следующей главе.

Листинг 24.4. Обработчик команд — реализация прикладной службы

```
public class CreateOrUpdateCategoryHandler
{
    // ...

    public ICommandResult Execute(CreateOrUpdateCategoryCommand command)
    {
        var category = new Category
        {
            CategoryId = command.CategoryId,
            Name = command.Name,
            Description = command.Description
        };
        if (category.CategoryId == 0)
            categoryRepository.Add(category);
        else
            categoryRepository.Update(category);
        unitOfWork.Commit();
        return new CommandResult(true);
    }
}
```

Так как сторона команд предметной модели предназначена для реализации предметных правил и логики, она не должна содержать свойств, необходимых пользовательскому интерфейсу. Философия DDD призывает сосредоточиться на обработке команд, а не на попытках смоделировать действительность. Обработчики помогают сконцентрировать внимание на поведении агрегатов и инвариантах, а не на окружающей действительности. Для выполнения специализированных команд не нужны всеобъемлющие предметные сущности (например, имя клиента не требуется для выполнения каких-либо операций с агрегатом). Сущность, пред-

ставляющая клиента, не имеет смысла. Осознание этого поможет сохранить агрегаты компактными, а значит, с меньшим числом зависимостей.

Запросы: информация о предметной модели

Сторона запроса, как можно видеть на рис. 24.5, решает задачи предоставления информации о предметной области. Объекты, возвращаемые стороной запросов, — это простые объекты DTO моделей представления, специализированные под потребности отображения информации. Предметная модель для стороны команд здесь не требуется, так как представление можно сгенерировать непосредственно на основе данных из хранилища. Сторона запросов не нуждается в абстракциях хранилищ, поэтому в данном контексте нет смысла использовать репозитории; здесь следует использовать фреймворки доступа к хранилищам или легковесные библиотеки, такие как ADO.NET.



Рис. 24.5. Сторона запросов в SQRS

В связи с тем, что модель чтения все еще находится в предметном уровне, в своих вычислениях она может использовать предметные объекты, если предварительно вычисленные данные отсутствуют в хранилище. Обычно для вычисления свойств моделей представления на основе данных, извлекаемых из хранилища, используются классы спецификаций.

Отображение отчетов непосредственно в модель данных

Сторона чтения в описываемой архитектуре отображает запрашиваемые отчеты непосредственно в модель данных, полностью минуя модель команд. Модели представлений конструируются внутри модели данных для каждой экранной формы или отчета. В результате получается множество готовых представлений, которые быстро извлекаются, просто отображаются в модели представления и поддерживают постраничное отображение, сортировку и поиск по произвольному тексту.

Если сторона команд не вычисляет предварительно какое-либо необходимое значение, используйте спецификацию или предметную службу для его вычисления «на лету», как показано в листинге 24.5.

Листинг 24.5. В запросах можно использовать предметные службы и спецификации

```
public class OrderQuery
{
    // ...

    public OrderViewModel Generate(Guid customerId)
    {
        var customer = _customerRepository.FindBy(customerId);

        var customerView = MapCustomerViewModelFrom(customer);

        customerView.IsInfluential =
            _influentialSpec.SatisfiedBy(customer);

        return customerView;
    }
}
```

Для обработки запросов можно также использовать инструменты объектно-реляционного отображения (Object Relational Mapper, ORM) — рекомендуется выбрать что-то легковесное, способное быстро отображать данные в объекты DTO модели представления. Благодаря этому сторона чтения получится простой — лишенной любой логики извлечения данных и отображения их в объекты DTO и делегирующей принятие решений на основе хранящихся данных спецификациям и предметным службам.

Материализованные представления на основе предметных событий

Можно пойти еще дальше по пути отделения операций чтения и записи, задействовав разные схемы данных. Модель чтения можно построить на основе предметных событий, генерируемых командами, если использовать их для создания материализованных представлений (materialized views), как показано на рис. 24.6.

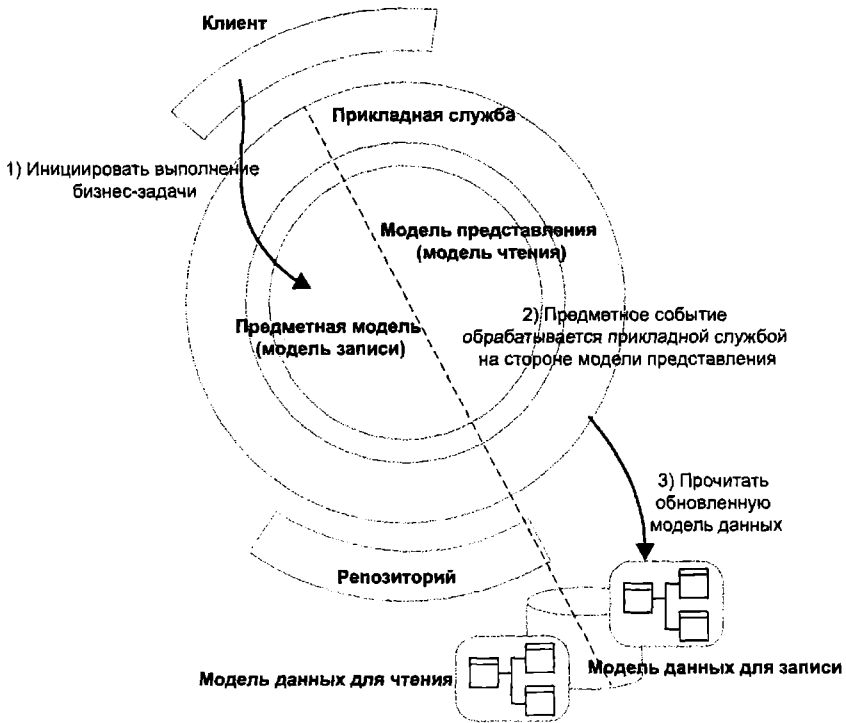


Рис. 24.6. Использование отдельного хранилища для чтения

Модель данных для чтения можно денормализовать и оптимизировать под запросы, включая предварительное вычисление данных.

В листинге 24.6 показано, как действия в модели команд приводят к возбуждению предметного события и последующему сохранению измененной модели данных для чтения. Все это происходит в рамках одной транзакции и одной базы данных. Далее вы увидите, как организовать обновление модели данных для чтения в другом процессе, чтобы повысить масштабируемость приложения.

Листинг 24.6. Обработчики команд могут управлять обновлением моделей представления

```
public class ModifyCategoryHandler
{
    public void Execute(ModifyCategoryCommand command)
    {
        var category = _catalogueRepository.FindBy(command.Id);

        using (DomainEvents.Register<CategoryUpdated>(onCategoryUpdated))
        {
            category.Update(command);
        }
    }
}
```

```
}  
  
private void onCategoryUpdated(CategoryUpdated @event)  
{  
    _catalogueViewModel.Update(CategoryUpdated);  
}  
}
```

Ошибочные представления о шаблоне CQRS

Существует множество ошибочных представлений о шаблоне CQRS, однако, как вы уже знаете, суть его заключается в использовании специализированной модели в каждом контексте. Если прежде вы читали какие-либо статьи о CQRS, у вас могло сложиться впечатление, что данный шаблон требует использования тяжелых фреймворков обмена сообщениями или реализации потенциальной непротиворечивости хранилища для чтения. В действительности это не так. Однако, как вы узнаете далее, эти методики можно использовать для расширения шаблона CQRS с целью повышения масштабируемости приложений. В данном разделе перечислены типичные ошибочные представления о шаблоне CQRS, с которыми вы можете встретиться.

Шаблон CQRS сложен в реализации

Если вы читали эту главу с самого начала, шаблон CQRS должен выглядеть для вас простым и удобным. По сути — это реализация принципа SRP на уровне предметной модели. Его удобно использовать для преодоления сложностей, возникающих, когда модель представления существенно отличается от оперативной модели. Шаблон CQRS не требует использования фреймворков, нескольких баз данных или шаблонов проектирования. Он лишь указывает, что для большей эффективности два разных контекста должны обслуживаться по отдельности. Концептуально это образ мышления, а не набор сложных шаблонов и принципов, которые необходимо применять.

Шаблон CQRS реализует потенциальную непротиворечивость

Потенциальная непротиворечивость — это практика асинхронного обновления модели чтения по отношению к транзакционной модели. Она не является обязательным требованием шаблона CQRS, но часто используется совместно с ним для повышения масштабируемости модели чтения. Потенциальная непротиворечивость моделей чтения привносит дополнительные сложности в приложение, так как пользователи, пытающиеся увидеть результаты своих действий, могут удивиться, увидев устаревшую информацию. Шаблон CQRS не требует потенциальной непротиворечивости. Вы можете использовать ту же базу данных и те же транзакции для обновления модели для чтения. Фактически хранилище для

чтения в вашем приложении уже может быть потенциально непротиворечивым, если активно пользоваться механизмами кэширования. Если вы решили применить шаблон CQRS, чтобы избавиться от сложностей, возникающих в результате размывания границ между задачами чтения и записи, попробуйте сначала использовать подход немедленной непротиворечивости и переходите к потенциальной непротиворечивости только при появлении проблем с производительностью. Поддержка потенциальной непротиворечивости для хранилища чтения требует дополнительных накладных расходов, о которых вы узнаете позже.

Модели должны поддерживать регистрацию событий

Как рассказывалось в главе 23, регистрация событий — весьма эффективная методика создания обеих моделей, чтения и записи; однако нет никаких предпосылок, требующих использования регистрации событий, как и предметных событий вообще, с шаблоном CQRS. Регистрация событий решает проблему точности учета происходящего, но этот прием действительно упрощает создание модели чтения, потому что позволяет создавать любые проекции, опираясь на хронологию событий.

Команды должны выполняться асинхронно

Шаблон CQRS не требует реализации команд по принципу «запустил и забыл». Применение асинхронных команд имеет смысл для большего удобства в моделях, когда множество пользователей может одновременно изменять одни и те же данные. Это позволяет обрабатывать их последовательно, масштабировать приложения и избегать критических нагрузок. Однако команды, не возвращающие подтверждения успеха или неудачи операции, должны иметь другие способы извещения пользователей о результатах их действий. Это может быть электронная почта или дополнительные функции, обрабатывающие сообщения об ошибках. В случае с покупками это может быть предложение вариантов замены клиенту вместо простого отказа при невозможности выполнить заказ.

Шаблон CQRS работает только с системами обмена сообщениями

Если вы ищете возможность реализовать потенциальную непротиворечивость хранилища для чтения или организовать асинхронное выполнение команд, тогда использование фреймворка обмена сообщениями, вероятно, можно считать неплохой идеей. В противном случае добавление поддержки сообщений в приложение лишь принесет ненужные сложности.

Шаблон CQRS требует использовать предметные события

Использование событий для создания материализованной модели чтения — эффективный способ разделения моделей чтения и записи; однако это не является обязательным условием, и вы можете использовать другие подходы к организа-

ции материализованных хранилищ для чтения. Как рассказывалось в главе 21 «Репозитории», состояние агрегатов можно получать с помощью шаблона «Хранитель». Для извлечения информации из предметных объектов с целью ее представления можно также использовать некоторые шаблоны из главы 26. Наконец, можно строить представления на основе реляционных моделей данных в моделях записи.

Шаблоны поддержки масштабируемости приложений

Шаблон CQRS помогает приложениям выдерживать большие нагрузки. Это достигается путем разделения сторон чтения и записи. Разделение сторон позволяет масштабировать их независимо, в соответствии с конкретными нуждами приложения. Данные для чтения и записи также могут храниться отдельно, если это лучше соответствует потребностям. В роли хранилища для записи, имеющего дело исключительно с агрегатами, можно использовать документоориентированную базу данных или другое хранилище, в котором данные организованы как пары «ключ — значение». В роли хранилища данных для чтения можно использовать реляционную базу данных или кэшированное хранилище.

Однако для масштабирования любой из сторон нужно знать и понимать связанные с этим достоинства и недостатки. Горизонтальное масштабирование — это не только техническое решение. Очень важно донести до заинтересованных лиц, что изменения в архитектуре системы повлекут изменения в восприятии пользователями, и такие изменения должны всесторонне оцениваться и быть приемлемыми для предприятия. Теорема CAP (также ее называют теоремой Брюера), которую мы обсуждали в главе 11 «Введение в интеграцию ограниченных контекстов», утверждает, что возможно одновременно обеспечить не более двух свойств из трех: непротиворечивость (consistency), доступность (availability) и устойчивость к разделению (partition tolerance). Поступаясь немедленной непротиворечивостью и переходя к использованию потенциальной непротиворечивости на стороне чтения, можно обеспечить улучшенную масштабируемость для приложений с существенными требованиями к представлению отчетов. Для приложений с большим количеством пользователей можно поступиться гарантией доступности и отказаться от немедленной обработки запросов. Для этого придется ставить в очередь сообщения, посылаемые в систему, обрабатывать их за пределами принимающего приложения и возвращать результаты обработки другими средствами, например по электронной почте или в виде отчетов по запросу.

Важно помнить, что любой компромисс в пользу масштабирования приложения влияет на восприятие пользователя и тем самым затрагивает интересы предприятия. Поэтому так важно, чтобы решения, влияющие на восприятие пользователей, принимали лица, ответственные за предприятие. В этом разделе будет рассмотрено множество способов масштабирования обеих сторон — записи и чтения — с использованием шаблона CQRS, вместе с компромиссами, которые необходимо учитывать.

Масштабирование стороны чтения: потенциально непротиворечивая модель чтения

Если к стороне чтения в приложении предъявляются более жесткие требования, чем к стороне записи, организация потенциально непротиворечивого хранилища для чтения поможет увеличить доступность и производительность приложения. Модель для чтения можно хранить в отдельной базе данных или копировать в несколько баз данных. Способ хранения данных для чтения может полностью отличаться от способа хранения состояния предметной модели на стороне записи. Однако в этом случае необходимо предусмотреть на стороне записи публикацию изменений, чтобы они могли быть денормализованы и сохранены для нужд чтения. На рис. 24.7 показано, что стороны записи и чтения связаны очередью. Эта очередь содержит предметные события, которые возбуждаются при изменении состояния предметной модели. Сторона чтения обрабатывает предметные события и обновляет данные для чтения. В результате, из-за небольшой рассинхронизации со стороной записи, она становится потенциально непротиворечивой.

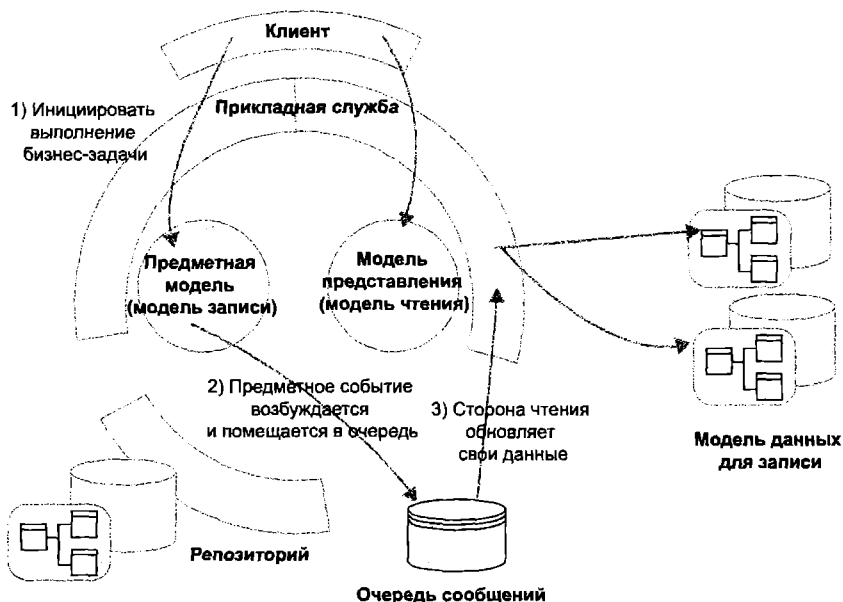


Рис. 24.7. Потенциально непротиворечивая модель чтения

Влияние на восприятие пользователя

Потенциальная непротиворечивость хранилища для чтения оказывает существенное влияние на восприятие пользователя. Свежесть отображаемых данных (в пользовательском интерфейсе и в традиционных отчетах) играет важную роль

для пользователей. Всегда следует четко обозначать, насколько устаревшими являются данные, чтобы пользователи могли учитывать это при принятии решений на основе наблюдаемых ими данных.

Использование модели чтения для объединения информации из нескольких ограниченных контекстов

С помощью модели чтения можно объединить несколько представлений для упрощения отображения информации. Информацию о состоянии предметных моделей, экспортируемую другими ограниченными контекстами посредством RESTful URL или систем сообщений, можно объединить и использовать в хранилище для чтения с целью удовлетворения потребностей составного пользовательского интерфейса. На рис. 24.8 показано, как могла бы выглядеть такая система.

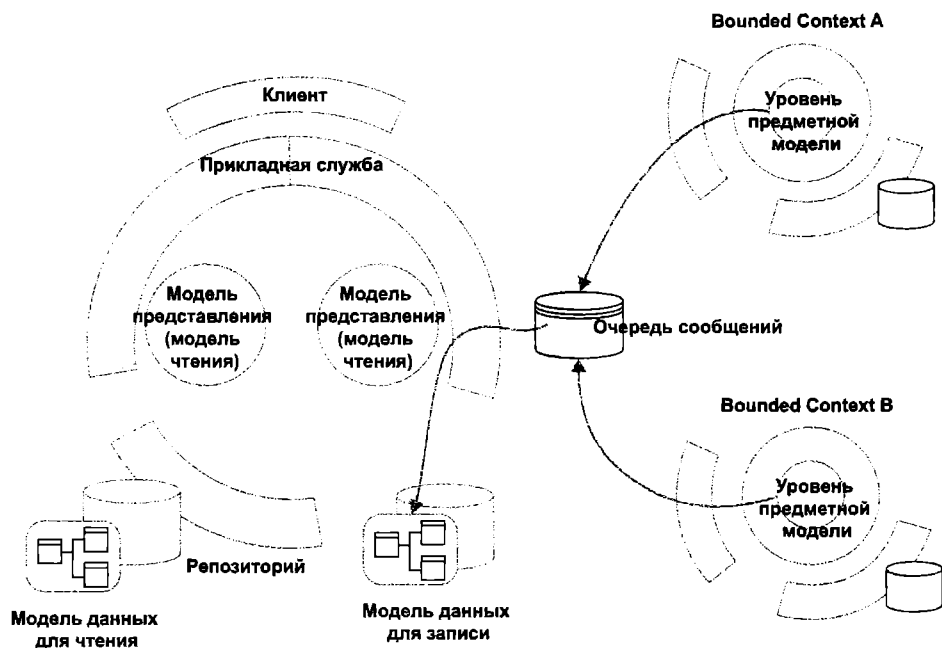


Рис. 24.8. Объединение данных из множества ограниченных контекстов в единой модели чтения

Базы данных отчетов или уровень кэширования

Возможно, вам уже приходилось использовать копию транзакционной базы данных в качестве базы данных для отчетов, которая создается на основе информации о доставке. Это одна из форм разделения задач чтения и записи. По возможности используйте этот простой метод масштабирования стороны чтения, как показано на рис. 24.9.

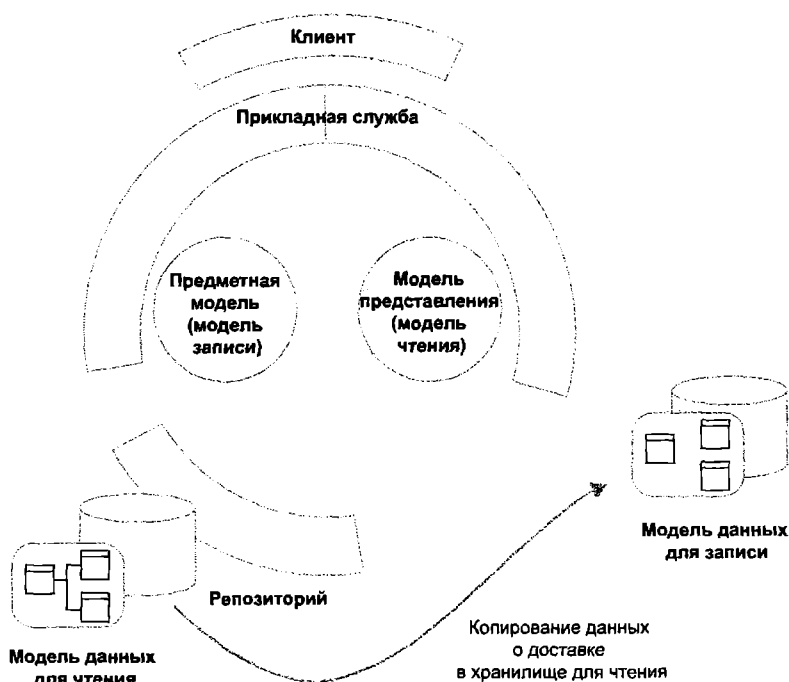


Рис. 24.9. Копирование транзакционной базы данных для модели чтения

Масштабирование стороны записи: асинхронные команды

Если предметная область предполагает поддержку большого числа пользователей, одновременно изменяющих один и тот же набор сущностей, возможность распределенного выполнения задач позволит масштабировать приложение и справиться с высокими нагрузками. На рис. 24.10 показано, как можно использовать очередь сообщений для сохранения запросов на выполнение бизнес-задач. Прикладной уровень принимает запрос от клиента на выполнение бизнес-задачи, но не приступает к ней немедленно, а передает для обработки другому процессу. Прикладной уровень может лишь подтвердить, что запрос получен; результат операции должен сообщаться каким-то другим способом, например посредством электронной почты или специальной страницы с информацией о состоянии запроса.

Валидация команд

Если запросы на выполнение бизнес-задач посылаются асинхронно, необходимо иметь некую гарантию их успешной обработки, так как не существует возможности немедленно получить подтверждение об успехе или неудаче операции. Прикладная служба должна выполнить некоторую простейшую валидацию запроса и, может быть, даже использовать хранилище представлений для проверки инва-

риантов перед добавлением запроса в очередь для последующей обработки. Приложение должно выполнить все необходимые проверки, чтобы гарантировать, что возможная неудача запроса будет обусловлена только лишь причинами, относящимися непосредственно к бизнесу, а не потому, что некоторый параметр отсутствует или имеет недопустимое значение.

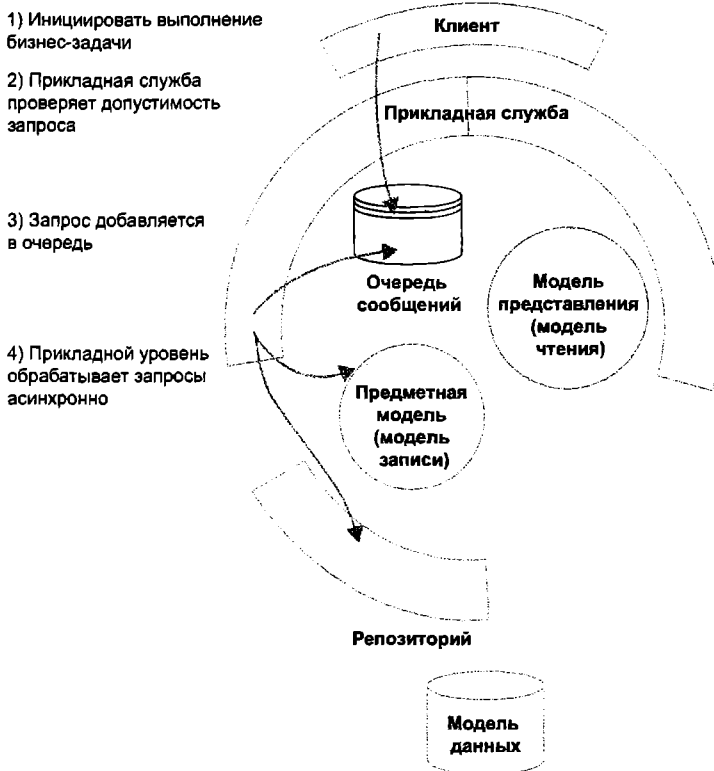


Рис. 24.10. Асинхронная работа стороны записи

Влияние на восприятие пользователя

Чтобы не вводить пользователей в заблуждение, следует четко сообщать, что обработка запроса осуществляется отдельно и данное подтверждение говорит не об успешной его обработке, а лишь о том, что он принят. В некоторых областях такой порядок работы, как размещение заказа на сайте электронной коммерции, знаком пользователям. Пользователя вполне устроит получение подтверждения об удовлетворении запроса, в котором содержится идентификационный номер созданного заказа. Он также понимает, что когда его заказ будет обрабатываться, может произойти ошибка при попытке списать средства с его банковской карты или некоторые товары могут закончиться на складе. Но если вы работаете в области, где пользователи привыкли сразу же видеть изменения, вызванные их запросами, необходимо четко сообщать им, что обработка запроса выполняется с задержкой.

Масштабирование всего вместе

Если предметная область предполагает обслуживание большого числа одновременных запросов чтения и записи, можно использовать потенциально непротиворечивую модель чтения в сочетании с асинхронным выполнением бизнес-задач. На рис. 24.11 показано, как могли бы выглядеть стороны чтения и записи, если бы потребовалось масштабировать их обе.

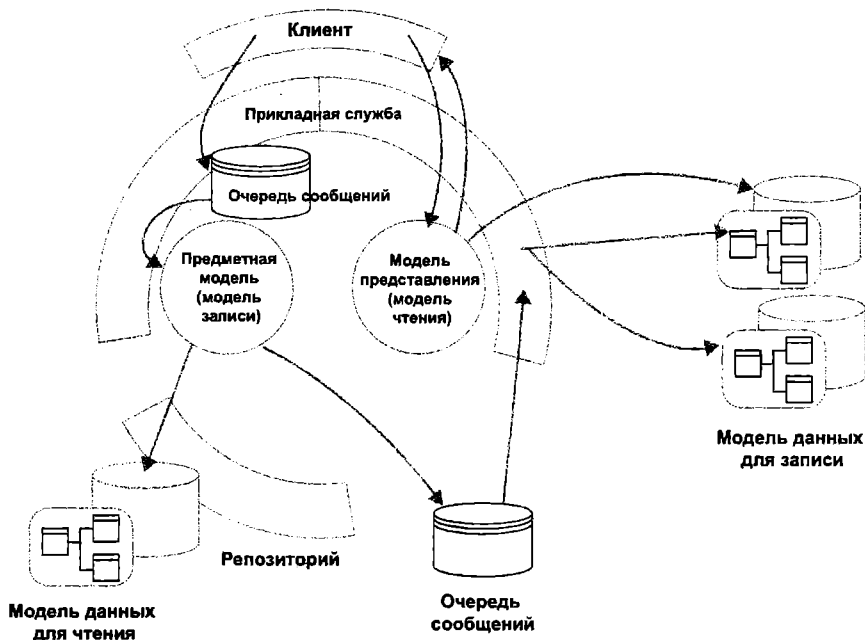


Рис. 24.11. Масштабирование сторон чтения и записи шаблона CQRS

Ключевые идеи

- CQRS — это не архитектурный шаблон и применяется не ко всем ограниченным контекстам в системе.
- Используйте шаблон CQRS, если предметная модель не способна отвечать потребностям представления и предметной логики без компромиссов. Разбейте модель на две специализированные модели: по одной для контекстов чтения и записи.
- Благодаря разделению агрегаты можно формировать, исходя исключительно из потребностей стороны записи и предметной логики, и проектировать их, опираясь на инварианты, а не на потребности представления информации.
- Модели представления на стороне чтения, специализированные под нужды отчетов, могут вообще не касаться предметной модели и извлекать информацию

непосредственно из базы данных, благодаря чему иметь лучшую производительность.

- CQRS — это простой шаблон, один из вариантов реализации принципа единственной ответственности (Single Responsibility Principle) на уровне модели, когда вместо одной модели создается две.
- CQRS часто неправильно рассматривают как применение практики обмена сообщениями, потенциальной непротиворечивости, предметных событий и приема регистрации событий. Несмотря на то что все перечисленное может пригодиться в определенных контекстах и использоваться для расширения системы, эти методики ни в коем случае не являются существенными для применения шаблона CQRS.
- CQRS позволяет масштабировать приложение при большом количестве операций чтения за счет ввода потенциальной непротиворечивости через разделение модели данных на две стороны — чтения и записи.
- При проектировании системы для области, где выполняется большое количество операций записи с одним и тем же множеством агрегатов, можно ввести асинхронную обработку запросов на стороне чтения.
- Масштабирование и введение потенциальной непротиворечивости в модели записи или чтения требует определенных компромиссов. Эти компромиссы должны быть понятны заинтересованным лицам и приняты ими, а также должны быть рассмотрены с позиции восприятия пользователей.
- CQRS позволяет системе пользоваться неограниченной масштабируемостью на обеих сторонах — чтения и записи.

25

Команды: шаблоны прикладных служб для обработки бизнес-сценариев использования

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Обсуждение различий между прикладной и предметной логикой
- Примеры задач, решаемых уровнем прикладных служб
- Шаблоны проектирования прикладных служб
- Рекомендации по тестированию прикладных служб

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке Download Code (Загружаемый код). Примеры кода для главы 25 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Многие выгоды от применения DDD являются результатом систематического использования единого языка (UL), как в общении со специалистами, так и в программном коде. Однако одной из наиболее серьезных проблем, с которыми при этом приходится сталкиваться, является поддержание четкой выраженности предметных понятий в коде за счет их отделения от решения исключительно технических задач. Например, описывая свою предметную модель во время переработки знаний со специалистами в предметной области, не загромождайте описание — или диалог — своими представлениями о потоках выполнения, сокетах или соединениях с базами данных. Старайтесь максимально ясно обрисовать свою предметную модель, четко отделив предметные понятия от исключительно технических проблем. Такое разделение играет важную роль в реализации прикладных служб.

Логически уровень прикладных служб располагается над предметным уровнем и зависит от него. Это означает, что главной задачей прикладных служб является координация выполнения сценариев использования с предметной моделью. Выполняя эту задачу, прикладные службы преобразуют входные и выходные данные для защиты предметной модели и часто нуждаются во взаимодействиях с другими ограниченными контекстами с использованием REST, сообщений и других

механизмов, которые обсуждались во второй части книги «Стратегические шаблоны: взаимодействия между ограниченными контекстами». На рис. 25.1 в общих чертах изображена роль прикладных служб.

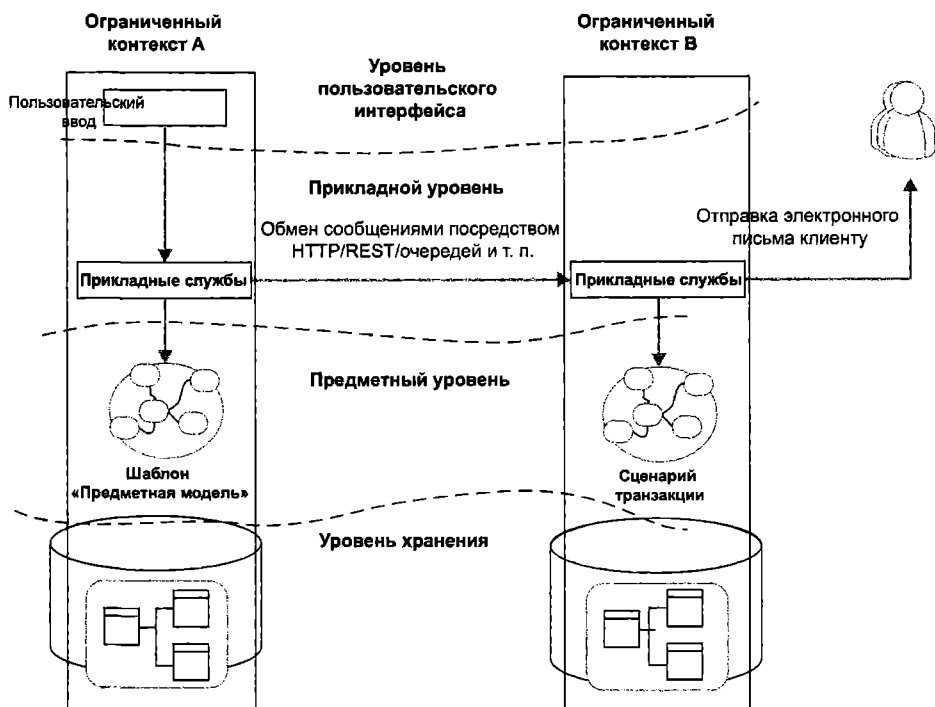


Рис. 25.1. Место и роль прикладных служб

Рисунок 25.1 устанавливает контекст для данной главы, наглядно показывая, как прикладные службы выполняют роль посредника между предметной моделью и внешними службами, такими как другие ограниченные контексты. Прежде чем перейти к техническим примерам, демонстрирующим назначение прикладных служб, важно уяснить различия между прикладной и предметной логикой. Даже опытные практики DDD иногда испытывают проблемы с таким различием.

Различение прикладной и предметной логики

Понимание различий между прикладной и предметной логикой совершенно необходимо для создания моделей, четко отражающих предметные понятия и изолирующих их от технических деталей. Часто это не очень сложная задача, однако могут возникнуть некоторые неясности. Целью этого раздела является прояснение картины, чтобы впоследствии вам реже приходилось сомневаться в выборе места для размещения программного кода.

Прикладная логика

Прежде всего, прикладные службы имеют две общие области ответственности. Во-первых, они отвечают за решение инфраструктурных задач: управление транзакциями, отправкой электронных писем и других похожих задач технического характера. Во-вторых, прикладные службы отвечают за координацию выполнения сценариев использования с предметной моделью. Правильное выполнение этих обязанностей помогает предотвратить «загрязнение» предметной логики техническими задачами или ошибочное ее размещение в прикладных службах.

Для демонстрации роли прикладных служб в этом разделе показано создание прикладной службы, которую можно использовать в приложении сетевой азартной игры. Она реализует сценарий использования «Приведи друга», в котором игрок получает на свой счет \$50, если у него получилось пригласить своего друга зарегистрироваться на игровом сайте. Друг также получает \$50 на счет при создании учетной записи. Кроме того, уровень лояльности того, кто привел друга, поднимается до уровня «золотой». Рисунок 25.2 иллюстрирует сценарий использования «Приведи друга».

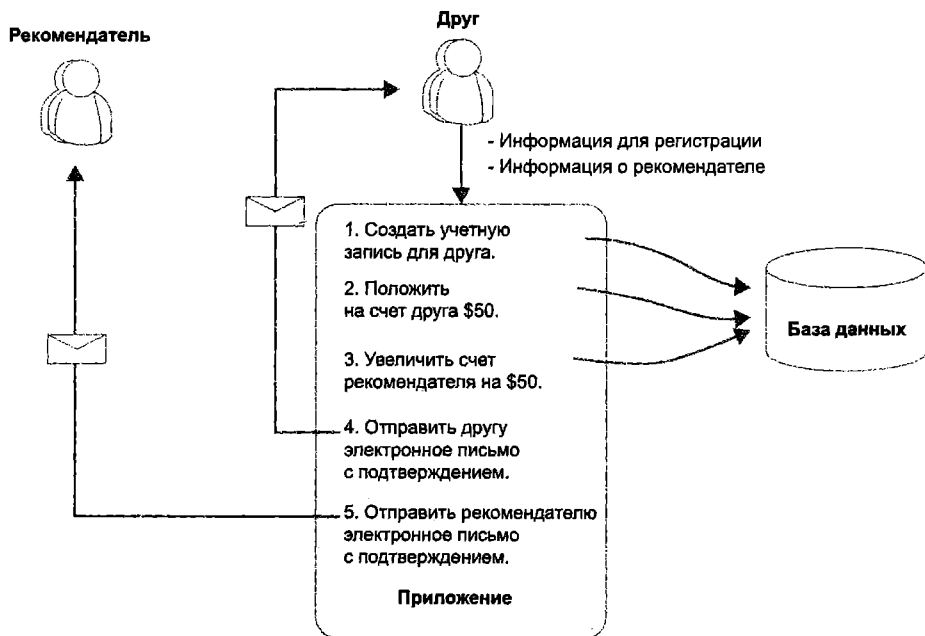


Рис. 25.2. Сценарий использования «Приведи друга»

Инфраструктурные задачи

Чтобы воспользоваться функциями предметной модели, необходимо решить множество инфраструктурных задач. Типичным примером может служить установка соединения с базой данных. Если вы приложили все силы, чтобы изолировать

такой инфраструктурный код, в награду вы получите более простую в сопровождении предметную модель, свободную от технических деталей. В следующих коротких разделах будут представлены примеры решения инфраструктурных задач в `RecommendAFriendService` — прикладной службе, ответственной за выполнение сценария использования «Приведи друга» на сайте азартных игр. Первое, что должно сделать данная служба, как и многие другие, — проверить входные данные.

ПРИМЕЧАНИЕ

Добавлять окончание «service» (служба) в имена прикладных служб совсем необязательно. В данной главе окончание «service» используется как помощь в обучении, но вы можете не использовать его в своих системах.

Проверка на прикладном уровне

Примерами проверок на прикладном уровне могут служить проверка допустимости типов входных параметров, правильности формата и длины. Они не являются бизнес-правилами, о которых можно узнать у специалистов, но могут служить причиной ошибочных условий в системе. Чтобы не загромождать предметную логику такими техническими деталями, проверки можно реализовать в прикладных службах.

Служба `RecommendAFriendService` принимает идентификатор учетной записи рекомендателя и информацию, необходимую для создания учетной записи друга. В листинге 25.1 представлена начальная реализация службы `RecommendAFriendService`, выполняющей проверку прикладного уровня.

Листинг 25.1. Проверка прикладного уровня в `RecommendAFriendService`

```
public class RecommendAFriendService
{
    public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
    {
        Validate(friendsAccountDetails);

        ...
    }

    // техническая проверка выполняется на прикладном уровне
    private void Validate(NewAccount account)
    {
        if (!account.Email.Contains("@"))
            throw new ValidationFailure("Not a valid email address");

        if (account.Email.Length >= 50)
            throw new ValidationFailure(
                "Email address must be less than 50 characters"
            );

        if (account.Nickname.Length >= 25)
            throw new ValidationFailure(
                "Nickname must be less than 25 characters"
            );

        if (String.IsNullOrEmpty(account.Email))
```

```

        throw new ValidationFailure("You must supply an email");

        if (String.IsNullOrEmpty(account.Nickname))
            throw new ValidationFailure("You must supply a Nickname");
    }
}

public class NewAccount
{
    public string Email { get; set; }

    public string Nickname { get; set; }

    public int Age { get; set; }
}

public class ValidationFailure : Exception
{
    public ValidationFailure(string message) : base(message) { }
}

```

Листинг 25.1 иллюстрирует роль проверки на прикладном уровне. Предложения в `Validate()` проверяют отдельные технические характеристики, такие как длина или формат строки с адресом электронной почты. Ни одно из них не является бизнес-правилом и ни одно из них не принадлежит предметной области.

Транзакции

В типичных бизнес-сценариях использования часто бывает необходимо выполнить несколько операций, обязательным условием выполнения которых должен быть либо успех, либо неудача для каждой из них. Осуществляя управление транзакциями в прикладных службах, вы получаете возможность гарантировать выполнение всех операций в предметной модели в рамках одной транзакции. Это можно продемонстрировать на примере службы `RecommendAFriendService`. Представьте, что руководством предприятия было решено не создавать новую учетную запись, если правило «Приведи друга» не может быть применено. То есть создание новой учетной записи и применение правила «Приведи друга» к обеим учетным записям должно происходить в рамках одной транзакции, как показано на рис. 25.3.

Добавить поддержку транзакций в `RecommendAFriendService` можно, воспользовавшись классом `TransactionScope` из .NET Framework, как показано в листинге 25.2. Не все базы данных, инструменты ORM и другие фреймворки используют `TransactionScope`, но прикладные программные интерфейсы (API) создания, подтверждения и прерывания транзакций обычно похожи.

Листинг 25.2. Управление транзакциями в `RecommendAFriendService`

```

public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    // большинство технологий имеет схожий программный интерфейс
    // управления транзакциями
    using (var transaction = new System.Transactions.TransactionScope())

```

```

{
    try
    {
        // ... многократные взаимодействия с предметной моделью
        transaction.Complete();
    }
    catch
    {
        // транзакция будет отменена, если Complete() не будет вызвана
    }
}
}

```

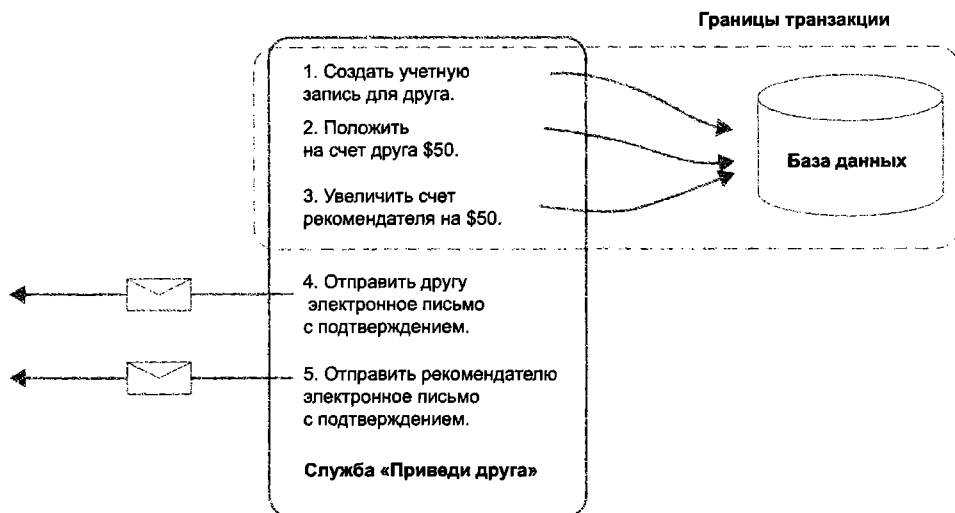


Рис. 25.3. Границы транзакции для сценария «Приведи друга»

Обработка и трансляция ошибок

Не все взаимодействия с предметной моделью будут успешными. В таких случаях, когда проверка в предметной модели терпит неудачу (даже при успешном прохождении проверок на прикладном уровне), почти наверняка будет возбуждено исключение или возвращен код ошибки. Задача прикладной службы состоит в том, чтобы обработать эти ошибочные условия и транслировать их в представление, подходящее для обработки внешней стороной, чтобы никакая внешняя сторона не была привязана к структуре предметных ошибок. Внешней стороной может быть человек, пользующийся веб-сайтом, или программная система.

Примером предметной ошибки в сценарии «Приведи друга» может служить ситуация, когда рекомендатель имеет непогашенную задолженность. Руководство предприятия приняло решение, что такие клиенты не должны вознаграждаться за лояльность. Поскольку предметная модель сообщает об этом ошибочном условии, служба `RecommendAFriendService` должна обработать ошибку и преобразовать ее в подходящее внешнее представление, как показано в листинге 25.3.

Листинг 25.3. Обработка и преобразование ошибок в `RecommendAFriendService`

```
public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    // большинство технологий имеет схожий программный интерфейс
    // управления транзакциями
    using (var transaction = new System.Transactions.TransactionScope())
    {
        try
        {
            // ... многократные взаимодействия с предметной моделью
            transaction.Complete();
        }
        catch(ReferralRejectedDueToLongTermOutstandingBalance)
        {
            throw new ApplicationError(
                "Sorry, this referral cannot be completed. The referrer " +
                "currently has an outstanding balance. Please contact " +
                "Customer Services"
            );
            // транзакция будет отменена, если Complete() не будет вызвана
        }
    }
}
```

ПРИМЕЧАНИЕ

Технически объект `TransactionScope` откатит транзакцию, когда его метод `Dispose()` будет вызван без вызова метода `Complete()`. В листинге 25.3 метод `Dispose()` вызывается автоматически в конце блока `using`.

Листинг 25.3 демонстрирует, как прикладные службы могут защищать структуру предметной модели, преобразуя исключения в стандартный формат. В данном примере внешним форматом является исключение `ApplicationError`. Использование исключения `ApplicationError` препятствует образованию зависимости клиентов прикладной службы от предметного исключения. Вместо этого образуется зависимость от исключения `ApplicationError` — более стабильного интерфейса, скрывающего потенциальную изменчивость предметной модели.

Использование соглашения по представлению ошибок, такого как исключение `ApplicationError`, дает возможность обрабатывать ошибки непротиворечивым способом. Например, для обработки исключений `ApplicationError` в приложении ASP.NET MVC можно создать класс, наследующий `HandleErrorAttribute`. Когда фильтр перехватит исключение `ApplicationError`, он будет знать, что исключение возбуждено прикладной службой и его можно безопасно преобразовать в сообщение для внешнего мира. Напротив, когда перехватывается исключение любого другого типа, фильтр не знает, можно ли безопасно извлечь информацию об ошибке, поэтому он возвращает обобщенное сообщение, как показано в листинге 25.4.

Листинг 25.4. Фильтр ошибок в ASP.NET MVC, использующий соглашения об обработке ошибок

```
public class ErrorFilter : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if (filterContext.Exception.GetType() == typeof(ApplicationError))
        {
            // вернуть конкретное сообщение для пользователя
            var msg = filterContext.Exception.Message;
            ErrorResponse(msg, filterContext);
        }
        else
        {
            // из соображений безопасности вернуть обобщенное сообщение
            var msg = "Sorry. Something really unexpected has occurred";
            ErrorResponse(msg, filterContext);
        }
    }

    public void ErrorResponse(string msg, ExceptionContext ec)
    {
        var routeData = new RouteValueDictionary(new { message = msg });
        var response = new RedirectToRouteResult("ErrorPage", routeData);
        ec.Result = response;
        ec.ExceptionHandled = true;
    }
}
```

Регистрация, оценка и мониторинг

Время отклика, ошибки и другая диагностическая информация помогает оценить работу приложения и выявить любые потенциальные проблемы на ранних стадиях. Но организация сбора этой информации может загромождать предметную логику и запутать важные предметные понятия. Иногда приходится просто терпеть этот беспорядок, но чаще для получения этой информации можно положиться на прикладные службы.

В листинге 25.5 показана дополненная версия `RecommendAFriendService`, регистрирующая продолжительность каждой операции и окончательный результат — исключение или успех, благодаря чему отпадает необходимость добавлять этот код в предметную модель.

Листинг 25.5. Регистрация и оценка в `RecommendAFriendService`

```
public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    using (var transaction = new System.Transactions.TransactionScope())
    {
        try
```

```

    {
        // ... многократные взаимодействия с предметной моделью
        transaction.Complete();

        // выполнить регистрацию здесь,
        // чтобы "не захламлять" предметную модель
        logger.Debug("Successful friend recommendation");
        StatsdClient.Metrics.Counter("friendReferrals");
    }
    catch (ReferralRejectedDueToLongTermOutstandingBalance ex)
    {
        // выполнить регистрацию здесь,
        // чтобы "не захламлять" предметную модель
        logger.Error(ex);
        StatsdClient.Metrics.Counter("ReferralRejected");
        throw new ApplicationError(
            "Sorry, this referral cannot be completed. The referrer " +
            "currently has an outstanding balance. Please contact " +
            "customer support"
        );
    }
}
}
}

```

Обратите внимание, как в листинге 25.5 используются методы `logger.Debug()`, `logger.Error()` и `StatsdClient.Metrics()`. Вызовы всех этих методов можно было бы внедрить в предметную логику, но вместо этого они были добавлены в прикладную службу.

Аутентификация и авторизация

Еще одной инфраструктурной задачей, которую приходится решать в большинстве приложений, является аутентификация. Для демонстрации аутентификации в сценарии «Приведи друга» в этом примере моделируется ситуация, когда сотрудник службы поддержки регистрируется в административном интерфейсе и вручную запускает процедуру обработки рекомендации, потому что возникла проблема с регистрацией нового клиента. В листинге 25.6 показано, как `AdminRecommendAFriendService` проверяет аутентичность пользователя перед запуском процедуры обработки рекомендации.

Листинг 25.6. Проверка аутентичности пользователя в `AdminRecommendAFriendService`

```

public class AdminRecommendAFriendService
{
    private IAuthenticationService authentication;

    ...

    public void RecommendAFriend(int referrerId, int friendId)
    {
        if (!authentication.IsLoggedInUser())

```

```
        throw new AuthenticationError();

    // поиск клиентов

    // запуск процедуры обработки рекомендации
}
}
```

Для многих приложений аутентификация не является достаточной мерой обеспечения безопасности, и требуется дополнительная авторизация. Авторизация — это процедура проверки наличия привилегий у пользователя, необходимых для выполнения запрошенной им операции. То есть авторизация — важный инструмент в приложениях, где разные пользователи имеют разные привилегии.

Представьте, что `AdminRecommendAFriendService` является частью приложения, с которым работает множество разных пользователей — клиентов и администраторов — и которое требует авторизации, чтобы не позволить простым пользователям запускать процедуру обработки рекомендации; если они смогут непрерывно наращивать свой баланс, просто взломав несколько адресов URL, это отрицательно скажется на бизнесе (такое действительно иногда случается). В листинге 25.7 представлена улучшенная версия `AdminRecommendAFriendService`, выполняющая проверку наличия привилегий администратора.

Листинг 25.7. Проверка привилегий, необходимых для запуска процедуры обработки рекомендации, в `AdminRecommendAFriendService`

```
public class AdminRecommendAFriendService
{
    private IAuthenticationService authentication;
    private IAuthorizationService authorization;

    // .. конструктор
    public void RecommendAFriend(int referrerId, int friendId)
    {
        if (authentication.IsLoggedInUser())
            throw new AuthenticationError();

        if (authorization.IsCurrentUserAdmin())
            throw new AuthorizationError();

        // запуск процедуры обработки рекомендации
    }
}
```

Обмен информацией

События, происходящие в одном ограниченном контексте, могут приводить к появлению событий, обрабатываемых другими ограниченными контекстами. Примеры этого были показаны во второй части книги, где демонстрировались приемы обмена сообщениями и использования архитектуры REST. За передачу событий между ограниченными контекстами отвечают прикладные службы. Для этого они могут публиковать события, используя шину сообщений (примеры можно найти

в главе 12 «Интеграция посредством обмена сообщениями»), ленту Atom (примеры можно найти в главе 13 «Интеграция с RPC и REST посредством HTTP») или другие средства обмена информацией. Важнейшей целью при этом является отделение предметных моделей от инфраструктурных задач. В листинге 25.8 показана дополненная версия `RecommendAFriendService`, публикующая события с использованием `NServiceBus`.

Листинг 25.8. Публикация событий в `RecommendAFriendService`

```
public class RecommendAFriendService
{
    private ICustomerDirectory customerDirectory;
    private IReferAFriendPolicy referAFriendPolicy;
    private IBus bus;

    public RecommendAFriendService(ICustomerDirectory customerDirectory,
                                   IReferAFriendPolicy referAFriendPolicy, IBus bus)
    {
        this.customerDirectory = customerDirectory;
        this.referAFriendPolicy = referAFriendPolicy;
        this.bus = bus;
    }

    public void RecommendAFriend(int referrerId,
                                 NewAccount friendsAccountDetails)
    {
        Validate(friendsAccountDetails);

        using (var transaction = new System.Transactions.TransactionScope())
        {
            try
            {
                var referrer = customerDirectory.Find(referrerId);
                var friend = customerDirectory.Add(friendsAccountDetails);
                referAFriendPolicy.Apply(referrer, friend);

                transaction.Complete();

                var msg = new CustomerRegisteredViaReferralPolicy
                {
                    ReferrerId = referrerId,
                    FriendId = friend.Id
                };
                bus.Publish(msg);
            }
            catch (ReferralRejectedDueToLongTermOutstandingBalance)
            {
                var msg = new ReferralSignupRejected
                {
                    ReferrerId = referrerId,
                    FriendEmail = friendsAccountDetails.Email,
                    Reason = "Referrer has long term outstanding balance"
                }
            }
        }
    }
}
```



```

        };
        bus.Publish(msg);
    }
}

private void Validate(NewAccount account)
{
    ...
}
}

```

Несмотря на то что в листинге 25.8 демонстрируется прикладная служба, использующая `NServiceBus`, обмен информацией с тем же успехом можно было бы реализовать с применением технологии вызова удаленных процедур (RPC), REST и пр. Прикладные службы должны также использоваться для решения других задач, описанных во второй части книги, включая обработку внешних событий, а также обработку и публикацию внутренних событий (по аналогии со шлюзом сообщений, как описывалось в главе 12) и производство ленты Atom.

Координация выполнения бизнес-сценариев использования

Взглянув еще раз на последнюю версию прикладной службы `RecommendAFriendService` в листинге 25.8, вы можете заметить, что она решает множество инфраструктурных задач, которые не добавляют ценности предприятию. Они необходимы только для поддержки второй важной области ответственности прикладных служб, добавляющей ценность, — координации выполнения сценариев использования с предметной моделью. В листинге 25.9 представлен фрагмент дополненной версии `RecommendAFriendService`, которая, опираясь на инфраструктурный фундамент, координирует выполнение сценария «Приведи друга» с предметной моделью.

Листинг 25.9. Координация выполнения сценария использования с предметной моделью в `RecommendAFriendService`

```

try
{
    // customerDirectory - предметный репозиторий
    Customer referrer = customerDirectory.Find(referrerId);
    Customer friend = customerDirectory.Add(friendsAccountDetails);

    // RecommendAFriendPolicy - реализация процедуры рекомендации
    RecommendAFriendPolicy.Apply(referrer, friend);

    transaction.Complete();

    // выполнить регистрацию событий здесь,
    // чтобы не загромождать предметную логику
    logger.Debug("Successful friend recommendation");
    StatsdClient.Metrics.Counter("friendReferrals");
}

```

Как видите, в листинге 25.9 используются переменные `customerDirectory` и `RecommendAFriendPolicy`. Они представляют предметные объекты и используются в данном примере, чтобы показать, как прикладная служба может многократно взаимодействовать с предметной моделью для выполнения сценария использования от начала до конца. Здесь она извлекает рекомендателя, предлагает предметной модели создать новую учетную запись и, наконец, требует от предметной модели выполнить процедуру обработки рекомендации.

Интеграция прикладных служб с фреймворками

Прикладные службы известны своей склонностью к раздуванию за счет слабо-связанной логики, которая, как может показаться, не принадлежит никакой другой области в приложении. Иногда некоторые фрагменты службы можно выделить в обособленные классы. Так, метод `Validate()` в службе `RecommendAFriendService` является отличным кандидатом на выделение в отдельный класс. Еще одной причиной раздувания прикладных служб является многократное повторение типового кода. В `RecommendAFriendService` таким типовым кодом является код управления транзакциями, который, в конечном счете, приходится использовать в каждой прикладной службе. Чтобы исключить эту причину, можно создать базовую прикладную службу, реализовав шаблон проектирования «Шаблонный метод», или написать утилиту, принимающую лямбда-выражение и обертывающую его транзакцией.

Однако иногда можно найти более ясное решение, нежели применение всеохватывающих прикладных служб, даже с использованием только что упомянутых шаблонов. Многие современные фреймворки предоставляют свои средства решения инфраструктурных задач. Отличным примером может служить ASP.NET MVC. В этом фреймворке поддерживается понятие фильтра операций `ActionFilter`, который действует как конвейер. Фильтры операций позволяют внедрять свои фильтры, осуществляющие проверки, открывающие и закрывающие транзакции, выполняющие журналирование и решающие другие инфраструктурные задачи. В листинге 25.10 показан пример класса `TransactionFilter`, наследующего `ActionFilter`.

Листинг 25.10. Фильтр операций для интеграции с фреймворком ASP.NET MVC

```
public class TransactionFilter : ActionFilterAttribute, IActionFilter
{
    // выполняется перед вызовом контроллера
    public override void OnResultExecuting(ResultExecutingContext filterContext)
    {
        // запустить транзакцию
        var t = new TransactionScope();
        HttpContext.Current.Items["transaction"] = t;
        base.OnResultExecuting(filterContext);
    }

    // выполняется после вызова контроллера
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // закрыть транзакцию, запущенную данным запросом
        var t = (TransactionScope)HttpContext.Current.Items["transaction"];
        t.Complete();
    }
}
```

```
        base.OnActionExecuted(filterContext);
    }
}
```

Когда решение инфраструктурных задач перекладывается на фреймворк, потребность в прикладных службах уменьшается. В результате может возникнуть соблазн поместить те несколько строк кода, которые отвечают за координацию, непосредственно в контроллер, как это сделано в листинге 25.11.

Листинг 25.11. Перемещение логики координации в контроллер

```
public class RecommendAFriendController : Controller
{
    private ICustomerDirectory directory;
    private IRecommendAFriendPolicy policy;

    public RecommendAFriendController(
        ICustomerDirectory customerDirectory, IRecommendAFriendPolicy policy)
    {
        this.directory = customerDirectory;
        this.policy = policy;
    }

    // все инфраструктурные задачи решаются фреймворком
    public ActionResult Index(int referrerId, NewAccount friend)
    {
        var referrer = directory.Find(referrerId);

        var newAcct = directory.Create(friend);
        policy.Apply(referrer, newAcct);

        return View();
    }
}
```

Однако, несмотря на привлекательность, такой подход имеет большой недостаток, о котором следует знать. Если прикладная служба используется в нескольких местах, например в веб-интерфейсе и в обычных клиентских приложениях, — использование всеобъемлющей прикладной службы поможет предотвратить дублирование инфраструктурной логики в обоих интерфейсах. В каждом конкретном проекте вам придется решить, насколько стоит связывать себя используемыми фреймворками.

Предметная логика с точки зрения прикладной службы

Многие практики DDD испытывают сложности с разграничением прикладной и предметной логики. Инфраструктурные задачи обычно легко идентифицируются, но логика координации, объединяющая составные части в единый сценарий использования, иногда может выглядеть противоречащей принципу «никаких бизнес-правил в прикладных службах». Если взглянуть еще раз на листинг 25.9, можно увидеть три взаимодействия с предметной моделью: извлечение рекомен-

дателя, запрос на создание новой учетной записи клиента и выполнение процедуры обработки рекомендации. Кому-то может показаться, что эта логика должна находиться в предметной модели.

Одним из способов определить принадлежность последовательности взаимодействий предметной модели является вопрос, который вам следует задать самому себе: «Всегда ли это должно происходить?» или: «Является ли эта последовательность шагов неразрывной?». В случае утвердительного ответа данную последовательность можно считать предметной логикой, поскольку составляющие ее шаги всегда должны выполняться вместе. Однако если шаги в последовательности можно переставлять и комбинировать разными способами, скорее всего, они не являются предметной логикой. В сценарии «Приведи друга» можно было бы утверждать, что процедура обработки рекомендаций всегда должна применяться только к существующему и новому клиенту (рекомендателю и другу рекомендателя). Из этого следует, что логика в прикладной службе фактически является бизнес-правилом, которое лучше реализовать в предметной службе. Однако иногда возникает необходимость применить процедуру к двум существующим учетным записям, если в процессе создания возникла какая-то ошибка. В этом случае создание новой учетной записи и применение процедуры обработки рекомендации никак не связаны, а это указывает на то, что логика имеет прикладной характер.

Признаком отсутствия утечек предметных понятий в прикладные службы является выразительность взаимодействий с предметной моделью. Примером может служить метод `RecommendAFriendPolicy.Apply()`. Реализация этого метода увеличивает счет каждой учетной записи на \$50 и поднимает уровень лояльности рекомендателя до «золотого». Если бы эти операции выполнялись в прикладной службе, они были бы не такими выразительными, что явно свидетельствовало бы о просачивании предметных понятий за пределы предметной модели.

Шаблоны прикладных служб

Исходя из своих предпочтений и контекста, внутри прикладных служб можно использовать самые разные шаблоны проектирования и принципы. В службе `RecommendAFriendService` из предыдущего раздела вы видели простой объектно-ориентированный код, написанный без особых изысков. Часто такое решение является самым простым и лучшим из возможных. Но иногда оно может оказаться сложным в сопровождении или образующим нежелательные тесные связи. Для решения этих проблем сообществом были придуманы шаблоны проектирования, описываемые далее.

Процессор команд

Шаблон «Процессор команд» (`command processor`) позволит избежать создания больших прикладных служб, решающих множество задач. Вместо этого для каждого случая использования достаточно будет создать команду и ее обработчик (процессор). В листинге 25.12 показана монолитная прикладная служба, которая, как можно видеть, решает множество задач.

Листинг 25.12. Потенциально монолитная прикладная служба

```
public class BloatedRecommendAFriendService
{
    public void RecommendAFriend(int referrerId, NewAccount friend)
    {
        ...
    }

    public void RecommendAFriendInDifferentCountry(int referrerId,
                                                    NewAccount friend)
    {
        ...
    }

    public void ReverseFriendReferral(int referrerId, int friendId)
    {
        ...
    }

    public void ReferAFriendWithoutLoyaltyBonus(int referrerId, NewAccount friend)
    {
        ...
    }

    // еще множество подобных методов
}
```

Прикладные службы, такие как `BloatedRecommendAFriendService` в листинге 25.12, могут доставить немало проблем. Реализации методов могут существенно изменяться и в каждом могут использоваться разные зависимости. Для приложений, использующих шаблон CQRS, особенно типична слабая связанность компонентов. Поэтому подобная прикладная служба может превратиться в серьезный источник трений. Для ослабления этих трений можно использовать шаблон «Процессор команд».

Чтобы создать альтернативную версию `BloatedRecommendAFriendService` на основе шаблона «Процессор команд», сначала нужно определить команду, выражающую намерение и содержащую всю необходимую информацию. Пример такой команды можно видеть в листинге 25.13.

Листинг 25.13. Команда `RecommendAFriend`

```
// команда, выражающая намерение
public class RecommendAFriend
{
    public int ReferrerId { get; set; }

    public NewAccount Friend { get; set; }
}
```

После создания команды можно приступать к обработчику. Интерфейс процессора команд `RecommendAFriend` показан в листинге 25.14.

Листинг 25.14. Интерфейс процессора команд `RecommendAFriendProcessor`

```
public interface IRecommendAFriendProcessor
{
    void Process(RecommendAFriend command);
}
```

Создание команды и процессора, как показано выше, обычно применяется при определении отдельных интерфейсов для каждого сценария использования, чтобы изолировать области ответственности и увеличить выразительность. Но многие практики DDD дополняют шаблон уровнем косвенности, ослабляющим зависимости и позволяющим составлять цепочки из процессоров команд. Возможность объединения в цепочки можно использовать для создания конвейеров, решающих инфраструктурные задачи, такие как проверка данных, поддержка транзакций и журналирование, чтобы изолировать управление действиями в предметной модели. Такая версия шаблона требует определить обобщенный интерфейс процессора, как показано в листинге 25.15.

Листинг 25.15. Обобщенный интерфейс процессора команд

```
public interface ICommandProcessor<T>
{
    void Process(T command);
}
```

Каждый процессор команд должен реализовать этот интерфейс, чтобы дать возможность включать его в цепочки. Сначала нужно создать обработчик, специализированный для конкретного сценария использования. В данном случае это мог бы быть `RecommendAFriendProcessor`, представленный в листинге 25.16.

Листинг 25.16. Процессор команд, специализированный для конкретного сценария использования

```
public class RecommendAFriendProcessor : ICommandProcessor<RecommendAFriend>
{
    public void Process(RecommendAFriend command)
    {
        Console.WriteLine("Processing RecommendAFriend command");
    }
}
```

Универсальный процессор команд, который можно применить в любом сценарии использования, также должен реализовать интерфейс `ICommandProcessor`. В листинге 25.17 демонстрируются примеры универсальных процессоров, осуществляющих журналирование и управление транзакциями.

Листинг 25.17. Универсальные процессоры, которые можно задействовать в любых сценариях использования

```
public class LoggingProcessor<T> : ICommandProcessor<T>
{
    private ICommandProcessor<T> nextLinkInChain;
```

```
public LoggingProcessor(ICommandProcessor<T> processor)
{
    this.nextLinkInChain = processor;
}

public void Process(T command)
{
    // вывести в журнал что-либо до передачи в конвейер
    nextLinkInChain.Process(command);
    // вывести в журнал что-либо после обработки конвейером
}
}

public class TransactionProcessor<T> : ICommandProcessor<T>
{
    private ICommandProcessor<T> nextLinkInChain;

    public TransactionProcessor(ICommandProcessor<T> processor)
    {
        this.nextLinkInChain = processor;
    }

    public void Process(T command)
    {
        // запустить транзакцию
        try
        {
            nextLinkInChain.Process(command);

            // подтвердить транзакцию
        }
        catch
        {
            // откатить транзакцию
        }
    }
}
```

Самое важное, на что следует обратить внимание в листинге 25.17: каждый процессор команд принимает другой процессор в параметре конструктора. Обработав команду, процессор вызывает процессор, переданный в параметре конструктора. По существу, он обертывает (или декорирует) «дочерний» процессор, образуя конвейер, который может быть сколь угодно длинным. Еще одна важная особенность находится внутри метода `Process()`. Процессор может выполнять действия до и после выполнения дочернего процессора. Как можно видеть в `TransactionProcessor`, он запускает транзакцию, вызывает дочерний процессор (который в свою очередь может вызвать другой дочерний процессор) и затем подтверждает или откатывает транзакцию в зависимости от возникновения исключения в дочернем процессоре.

ПРИМЕЧАНИЕ

Процессор `TransactionProcessor` в листинге 25.17 содержит блок `try/catch`. Иногда бывает желательно вынести этот блок в отдельный процессор, реализующий обработку ошибок.

Связывание процессоров команд в цепочки — последняя оставшаяся особенность. Простейший способ конструирования таких цепочек показан в листинге 25.18. Сопоставление этого листинга с листингом 25.17 должно помочь понять, как создается и действует конвейер.

Листинг 25.18. Связывание процессоров команд для формирования конвейера

```
public static class Bootstrap
{
    public static ICommandProcessor<RecommendAFriend>
        RecommendAFriendProcessor { get; set; }

    public static void ConfigureApplication()
    {
        // создать внутренний процессор
        var RecommendAFriendProcessor = new RecommendAFriendProcessor();

        // обернуть внутренний процессор процессором журналирования
        var loggingProcessor =
            new LoggingProcessor<RecommendAFriend>(RecommendAFriendProcessor);

        // обернуть процессор журналирования (который обертывает
        // внутренний процессор) процессором транзакций
        var transactionProcessor =
            new TransactionProcessor<RecommendAFriend>(loggingProcessor);

        RecommendAFriendProcessor = transactionProcessor;

        // как вариант, можно использовать механизм внедрения зависимостей
    }
}
```

Существуют также другие способы конструирования конвейеров, отличные от представленного в листинге 25.18. Некоторые разработчики предпочитают вручную конструировать каждый конвейер способом, показанным выше, используя вспомогательные методы, чтобы избежать дублирования кода. Другие отдают предпочтение внедрению зависимостей. В конечном счете, вам придется самостоятельно решать, как создавать свои конвейеры.

Публикация/подписка

Шаблон «Публикация/подписка» (`publish/subscribe`) помогает ослабить связи за счет подписки прикладных служб на предметные события. Этот шаблон очень удобен, когда предметная логика изначально основана на событиях, особенно когда предметная модель не возвращает ответа на команды. Интерфейс версии событийно-ориентированной службы `IReferralPolicy` показан в листинге 25.19.

Листинг 25.19. Интерфейс для службы, основанной на событиях

```
public interface IReferralPolicy
{
    event EventHandler<Referral> ReferralAccepted;
    event EventHandler<Referral> ReferralRejected;
    void Apply(RecommendAFriend command);
}
```

Прикладная служба, реализующая интерфейс `IReferralPolicy` из листинга 25.19, передает команды методу `Apply()` как обычно. Однако чтобы узнать, была ли команда успешно обработана, служба подписывается на два события: `ReferralAccepted` и `ReferralRejected`. Реализация этого шаблона показана в листинге 25.20.

Листинг 25.20. Подписка на предметные события

```
public class RecommendAFriendService
{
    private IReferralPolicy policy;

    public RecommendAFriendService(Domain.IReferralPolicy policy)
    {
        // подписаться на предметные события
        policy.ReferralAccepted += HandleReferralAccepted;
        policy.ReferralRejected += HandleReferralRejected;

        this.policy = policy;
    }

    private void HandleReferralAccepted(object sender, Domain.Referral e)
    {
        // послать подтверждение, например, по электронной почте
    }

    private void HandleReferralRejected(object sender, Domain.Referral e)
    {
        // послать уведомление об отказе, например, по электронной почте
    }

    public void RecommendAFriend(int referrerId, NewAccount friend)
    {
        var command = new RecommendAFriend
        {
            ReferrerId = referrerId,
            Friend = friend
        };

        policy.Apply(command);
    }
}
```

Подписка на события, определяемые интерфейсом `IReferralPolicy`, происходит в конструкторе `RecommendAFriendService`, как показано в листинге 25.20. Всякий

раз, когда предметная модель посылает любое из этих событий, вызывается соответствующий обработчик: `HandleReferralAccepted()` или `HandleReferralRejected()`. Эти события публикуются в ответ на команды, передаваемые в предметную модель методом `RecommendAFriend()`.

Проблема решения в листинге 25.20 заключается в невозможности корректной обработки транзакций, потому что обработчики событий не имеют доступа к объекту транзакции и, соответственно, не могут ни подтвердить, ни откатить транзакцию. Чтобы исправить эту проблему, можно определить поле экземпляра для хранения объекта транзакции. Однако в этом случае вам придется обеспечить создание нового экземпляра `RecommendAFriendService` для каждой новой транзакции, чтобы избежать проблем, характерных для многопоточного выполнения.

Еще одна проблема примера в листинге 25.20, связанная с транзакциями, проявляется при выполнении в многопоточной среде. Представьте, что команда посылается асинхронно, из другого потока выполнения с применением механизма задач в C#:

```
Task.Factory.StartNew(() => policy.Apply(command));
```

Когда метод `Apply()` вызывается асинхронно, он оказывается за пределами области видимости охватывающей транзакции. К счастью, класс `TransactionScope` в .NET поддерживает режимы, упрощающие реализацию асинхронных сценариев (<http://stackoverflow.com/questions/13543254/get-transactionscope-to-work-with-async-await>). Но даже в этом случае при использовании транзакций в многопоточной среде следует проявлять осторожность. Кроме того, возможно, вас заинтересует шаблон, основанный на использовании ключевых слов `async/await`.

Шаблон запрос/ответ

Перед выбором более сложных шаблонов проектирования всегда следует рассмотреть возможность применения простого шаблона «Запрос/ответ» (`request/reply`). Этот шаблон следует принципу «одна модель на входе — одна на выходе» (`One Model In One Model Out`, `OMIOMO`), когда прикладная служба принимает один объект переноса данных (`DTO`) и возвращает один `DTO`, как демонстрирует листинг 25.21.

Листинг 25.21. Прикладная служба, реализующая шаблон «Запрос/ответ»

```
public class RecommendAFriendService
{
    private IReferralPolicy policy;

    public RecommendAFriendService(Domain.IReferralPolicy policy)
    {
        this.policy = policy;
    }

    public RecommendAFriendResponse RecommendAFriend(
        RecommendAFriendRequest request)
    {

```

```
try
{
    var command = new RecommendAFriend
    {
        ReferrerId = request.ReferrerId,
        Friend = request.Friend
    };

    policy.Apply(command);

    return new RecommendAFriendResponse
    {
        Status = RecommendAFriendStatus.Success
    };
}
catch (ReferralRejectedDueToLongTermOutstandingBalance)
{
    return new RecommendAFriendResponse
    {
        Status = RecommendAFriendStatus.ReferralRejected
    };
}
}
```

Типичным соглашением является использование окончания «request» в названии типа входной модели и окончания «response» в названии типа выходной модели. Помимо этого необязательного соглашения, оба объекта должны быть простыми объектами DTO, не содержащими предметных объектов, и оба должны определяться на уровне прикладной службы. В листинге 25.22 показан объект DTO `RecommendAFriendRequest`, а в листинге 25.23 — объект DTO `RecommendAFriendResponse`.

Листинг 25.22. DTO-модель запроса

```
public class RecommendAFriendRequest
{
    public int ReferrerId { get; set; }

    public NewAccount Friend { get; set; }
}
```

Листинг 25.23. DTO-модель ответа

```
public class RecommendAFriendResponse
{
    public RecommendAFriendStatus Status { get; set; }
}

public enum RecommendAFriendStatus
{
    Success,
    ReferralRejected
}
```

Еще одним типичным соглашением при использовании шаблона «Запрос/ответ» является включение в объект ответа информации о результатах выполнения бизнес-сценария использования. Так, в случае возникновения ошибки или отвержения процедуры вместо возбуждения исключения в объекте ответа прикладной службе передается соответствующая информация о состоянии.

Вообще говоря, шаблон «Запрос/ответ» — один из простейших шаблонов, опирающихся на процедурный программный код. Если нет нужды в преимуществах других, более сложных шаблонов, лучше сохранить решение максимально простым и упростить жизнь другим разработчикам, применяя шаблон «Запрос/ответ», пока это не начнет вызывать сложности.

async/await

Разработчики на C# могут писать однопоточный программный код, выполняющий асинхронные операции с применением ключевых слов `async` и `await`, добавленных в C# 5. Вам определенно стоит учитывать возможность их использования при создании асинхронных, неблокирующих приложений. Однако этот шаблон может не соответствовать вашим требованиям к ясности и выразительности предметной модели из-за того, что асинхронные методы должны возвращать значение типа `Task<T>`. Эта проблема демонстрируется в листинге 25.24.

Листинг 25.24. Репозиторий, совместимый с `async/await`

```
public interface ICustomerDirectory
{
    Task<Customer> Find(int customerId);

    Task<Customer> Create(NewAccount details);
}
```

В листинге 25.24 можно видеть, что представление каталога клиентов возвращает `Task<Customer>`. Сделано это с тем, чтобы реализация могла использовать неблокирующие вызовы к базе данных, более эффективные и масштабируемые. Очевидно, что это несколько «загрязняет» предметную модель. Однако выразительность `async` и `await` смягчает этот недостаток, как показывает реализация `RecommendAFriendService` в листинге 25.25.

Листинг 25.25. `RecommendAFriendService`, совместимая с `async/await`

```
public async void RecommendAFriend(int referrerId, NewAccount friend)
{
    // ...
    var referrer = await directory.Find(referrerId);
    var newAcct = await directory.Create(friend);
    policy.Apply(referrer, newAcct);
    // ...
}
```

Одна из выгод выполнения трех вызовов в `RecommendAFriend()`, а не в предметной модели состоит в том, что технические детали `async/await` не загромождают предметную логику, как иллюстрирует листинг 25.25. Отметьте также, что метод

`RecommendAFriend()` в листинге 25.25 отмечен ключевым словом `async`. Это гарантирует, что после вызова метод будет выполняться асинхронно, если содержит асинхронные вызовы. Две первые строки в теле метода — это фактически асинхронные вызовы; на это указывает ключевое слово `await`. Когда среда выполнения .NET достигает ключевого слова `await`, она пытается избежать блокировки потока выполнения до окончания вызова. Используя это решение, необходимо проявлять особую осторожность при работе с транзакциями (<http://stackoverflow.com/questions/13543254/get-transactionscope-to-work-with-async-await>).

Взвесьте все «за» и «против» перед использованием `async` и `await`. Их применение поможет более эффективно использовать ресурсы, но они добавляют лишние помехи в код. К счастью, как показывает листинг 25.25, большая часть помех оказывается в прикладной службе, а не в предметной модели. Этот подход может оказаться вполне приемлемым компромиссом для ваших проектов, чтобы стремиться к нему.

ПРИМЕЧАНИЕ

Дополнительную информацию об асинхронных методах в ADO.NET, совместимых с `async/await`, можно найти в MSDN (<http://blogs.msdn.com/b/adonet/archive/2012/07/15/using-sqldatareader-s-new-async-methods-in-net-4-5-beta-part-2-examples.aspx>). Также в блоге Джона Скита можно познакомиться с реализацией `async/await` на основе конечного автомата (Jon Skeet) (<http://codeblog.jonskeet.uk/2011/05/08/eduasync-part-1-introduction/>).

Тестирование прикладных служб

Осуществляя тестирование на уровне прикладных служб, можно охватить значительные вертикальные срезы функциональных возможностей. В идеале желательно охватить тестированием как можно больший объем кода в условиях, максимально приближенных к эксплуатационным. Такие тесты могут оказаться в разных группах, включая системные, прямо-сдаточные, интеграционные и функциональные тесты.

ПРИМЕЧАНИЕ

Системные, прямо-сдаточные и функциональные тесты можно выполнять на более высоком уровне действующего экземпляра приложения. Типичными примерами могут служить тестирование HTTP-ответов (для проверки API) и автоматизированное тестирование веб-страниц в браузере.

Используйте предметную терминологию

Тесты могут быть фантастической возможностью выразить предметные понятия, если реализовать их на едином языке. Возможно, вам понадобится дополнительно пообщаться со специалистами в предметной области, чтобы создать тесты, опираясь на их требования и используя приемы разработки через реализацию поведения. Это даст вам возможность совместно со специалистами проверить соответствие формулировок тестов предметным понятиям.

В листинге 25.26 приводится схематическая реализация тестов для проверки сценария «Приведи друга» на уровне прикладной службы. Обратите внимание, как названия тестов отражают предметные понятия.

Листинг 25.26. Схематическая реализация высокоуровневых тестов, выраженных на едином языке проекта

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Tests
{
    [TestClass]
    // имя класса отражает бизнес-сценарий использования
    public class Recommend_a_friend
    {
        // этот метод будет вызван первым (и только один раз),
        // вслед за ним будут вызваны остальные методы
        [ClassInitialize]
        public void When_a_user_signs_up_with_a_referral_from_a_friend()
        {
            ...
        }

        [TestMethod]
        public void The_referrer_has_50_dollars_credited_to_their_account()
        {
            ...
        }

        [TestMethod]
        public void The_friend_has_an_account_created_with_an_initial_50_dollars()
        {
            ...
        }

        [TestMethod]
        public void The_referrers_loyalty_is_upgraded_to_gold_status()
        {
            ...
        }

        [TestMethod]
        public void The_referrer_gets_an_email_notifying_of_the_referral()
        {
            ...
        }

        [TestMethod]
        public void The_friend_gets_an_email_notifying_of_account_creation()
        {
            ...
        }
    }
}
```

Тестируйте как можно больше функциональных возможностей

Чем больше тестов вы подготовите, тем крепче будет ваша уверенность в успешном и стабильном функционировании приложения после развертывания. Для этого нужно избегать использования подставных объектов-имитаций и заглушек, отдавая предпочтение использованию конкретных реализаций. В листинге 25.27 показано, как этот подход можно реализовать в тестовом классе `Recommend_a_friend`.

Листинг 25.27. Тестирование с использованием конкретных реализаций

```
[ClassInitialize]
public void When_a_user_signs_up_with_a_referral_from_a_friend()
{
    // тестировать как можно больше функциональных возможностей
    directory = new CustomerDirectory(new InMemoryDatabase());
    var policy = new ReferralPolicy();

    // отправку извещений по электронной почте нельзя протестировать,
    // поэтому используется заглушка
    emailer = MockRepository.GenerateStub<IEmailer>();

    service = new RecommendAFriendService(directory, policy, emailer);

    service.RecommendAFriend(referrerId, friendsDetails);
}
```

В листинге 25.27 конструируется служба `RecommendAFriendService`, которой передается конкретная реализация `CustomerDirectory`. Эта же реализация будет использоваться после развертывания приложения. Данный тест проверяет интеграцию репозитория `CustomerDirectory` и прикладной службы `RecommendAFriendService`.

В листинге 25.27 можно также заметить, что `CustomerDirectory` конструируется для использования с базой данных в памяти. Дело в том, что нельзя легко и быстро выполнить тестирование с действующей базой данных. Поэтому вместо действующей базы данных используется ее версия в памяти, которую легко настроить и которая позволяет быстро выполнить тестирование. Однако эти условия далеки от эксплуатационных, поэтому остается небольшая вероятность, что тест будет выполняться без ошибок, а при использовании действующей базы данных будут возникать проблемы. Добавление в набор нескольких сквозных тестов, которые используют действующую базу данных, поможет вам получить дополнительную уверенность.

Другой компонент, работу которого проверить не так легко, — объект `emailer`, поэтому фактическая реализация замещена фиктивной реализацией из фреймворка `RhinoMocks`. То есть данный компонент остается не охваченным тестированием; тест будет завершаться успешно, пока методы `emailer` вызываются с правильными аргументами, как показано в листинге 25.28.

Листинг 25.28. Проверка вызовов с помощью фиктивной реализации, когда фактическая не может быть протестирована

```
[TestMethod]
public void The_referrer_gets_an_email_notifying_of_the_referral()
{
}
```

```
var referrer = directory.Find(referrerId);
emailer.AssertWasCalled(em =>
{
    em.SendReferralAcknowledgement(referrer);
}));
}
```

Фреймворк RhinoMocks предоставляет метод `AssertWasCalled()`, который возбуждает исключение, если переданное ему лямбда-выражение не было вызвано в процессе выполнения теста. Поэтому в листинге 25.28 RhinoMocks возбудит исключение, если метод `emailer.SendReferralAcknowledgement()` не будет вызван с аргументом, представляющим рекомендателя. Однако, как отмечалось выше, объекты-имитации и заглушки часто являются последним средством, когда невозможно протестировать действительную реализацию. Вовлекая в тестирование фактическую реализацию, можно напрямую проверить соответствие результатов ожиданиям, как показано в листинге 25.29.

Листинг 25.29. Тестирование фактической реализации без применения имитаций и заглушек

```
[TestMethod]
public void The_referrers_loyalty_is_upgraded_to_gold_status()
{
    var referrer = directory.Find(referrerId);
    Assert.AreEqual(LoyaltyStatus.Gold, referrer.LoyaltyStatus);
}
```

Ключевые идеи

- Прикладные службы позволяют изолировать предметную логику от решения технических задач.
- К числу технических задач относятся, например, управление транзакциями и соединениями с базами данных, а также отправка уведомлений по электронной почте.
- Прикладные службы отвечают за координацию выполнения сценариев использования с предметными моделями.
- Взаимодействуя с предметной моделью, прикладная служба должна пользоваться высокоуровневым, выразительным программным интерфейсом предметных объектов.
- Важнейшей ответственностью прикладных служб является защита структуры предметных моделей посредством предоставления в распоряжение верхних уровней и внешних компонентов более стабильных интерфейсов для взаимодействий.
- На уровне прикладных служб можно использовать такие шаблоны, как «Процессор команд», а также шаблоны асинхронных взаимодействий.
- Тестирование прикладных служб дает возможность выразить высокоуровневое поведение или даже полные сценарии использования на едином языке, охватывая значительную долю реализации.

26

Запросы: предметная отчетность

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Рекомендации по созданию отчетов, не зависящих от структуры предметной модели
- Создание отчетов с использованием существующих предметных служб
- Создание отчетов в обход предметной модели, путем прямого обращения к базе данных
- Создание отчетов в приложениях, использующих прием регистрации событий
- Обсуждение компромиссов при выборе между отчетами, принадлежащими ограниченному контексту, и отчетами на основе данных из множества ограниченных контекстов

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу www.wrox.com/go/domaindrivendesign на вкладке **Download Code** (Загружаемый код). Примеры кода для главы 26 доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Программные системы создаются для удовлетворения нужд предприятий. В число этих потребностей входят не только функциональные возможности, приносящие прибыль, но и возможность оценки деятельности предприятия. Эта роль отводится отчетам: отслеживаются важные характеристики и ключевые показатели эффективности работы (Key Performance Indicators, KPI), такие как объем продаж, финансовые показатели и удовлетворенность клиентов. Как было описано ранее, существует множество способов создания систем с применением приемов предметно-ориентированного проектирования. Аналогично, существует множество способов реализации отчетов.

При выборе способов реализации отчетов в приложениях необходимо учитывать известные достоинства и недостатки: скорость разработки, простоту сопровожде-

ния, производительность и даже масштабируемость. В каких-то случаях будет достаточно просто создать новую веб-страницу, повторно использующую уже имеющийся программный код. В иных же, когда имеется множество распределенных ограниченных контекстов, может понадобиться создать отдельный ограниченный контекст для отчетов, который подписывается на события от других ограниченных контекстов и хранит всю информацию локально. Цель этой главы состоит в том, чтобы показать различные возможности, рассказать об их достоинствах и недостатках и снабдить вас всей необходимой информацией для принятия обоснованных решений в проектах, над которыми вы работаете.

ПРИМЕЧАНИЕ

В контексте этой главы под отчетом понимается представление связанного набора данных для некоторых аналитических целей. Фактически это означает лишь то, что отчет принимает на себя универсальную роль по представлению информации пользователям, а не является реализацией часто используемого формального понятия отчетов, содержащих коммерческую или оперативно-управленческую информацию. Например, сводная страница с общедоступной информацией об учетной записи в Твиттере, где отображается число последователей, число твитов (сообщений) и ретвитов (публикаций чужих сообщений), подпадает под более широкое определение отчетов, используемое в этой главе.

Предметная отчетность внутри ограниченного контекста

Наиболее простым способом реализации отчетов является тот случай, когда все необходимые данные находятся в пределах одного ограниченного контекста. Это также является свидетельством того, что вы правильно определили границы своих ограниченных контекстов, так как отчеты обычно предназначаются для использования определенными отделами на предприятии, каждому из которых соответствует единственный ограниченный контекст. Одно из важнейших решений, которое требуется принимать в ситуациях, когда не приходится беспокоиться о распределенных ограниченных контекстах, — использование предметного кода для создания отчетов.

Создание отчетов на основе предметных объектов

Для создания веб-страницы с отчетами можно использовать существующий предметный код, конструирующий модель представления. В случае необходимости быстро создать страницу это обычно самый первый рассматриваемый вариант, так как для его реализации требуется меньше всего усилий. Самым большим его недостатком является невысокая производительность, потому что при использовании этого варианта вы не имеете возможности управлять запросами к хранилищу данных. Если производительность не так важна, как скорость разработки, этот вариант может оказаться неплохим выбором.

В следующих примерах вы узнаете, как создать отчет с показателями работы дилеров. Сначала будет представлен пример, использующий простые отображения,

а затем вы увидите альтернативную реализацию с использованием шаблона проектирования «Посредник» (mediator).

С использованием простых отображений

Самый быстрый способ создать отчет — это, пожалуй, взять предметные объекты и отобразить их свойства в модель представления, которая определяет логику и данные, необходимые для создания представления. Имея предметные объекты из листинга 26.1, которые уже используются в других частях приложения, можно легко заполнить модель представления в листинге 26.2, просто отображая свойства, как показано в листинге 26.3.

Листинг 26.1. Предметные сущности, хранящие данные для отчета с показателями работы дилеров

```
public interface IDealershipRepository
{
    Dealership Get(int dealershipId);
}

public interface IDealershipRevenueCalculator
{
    DealershipPerformanceActuals CalculateFor(Dealership dealership,
                                              DateTime start, DateTime end);
}

public interface IDealershipPerformanceTargetsProvider
{
    DealershipPerformanceTargets Get(Dealership dealership,
                                     DateTime start, DateTime end);
}

public class DealershipPerformanceTargets
{
    public int TargetRevenue { get; set; }

    public int TargetProfit { get; set; }
}

public class DealershipPerformanceActuals
{
    public int TotalRevenue { get; set; }

    public int NetProfit { get; set; }
}

public class Dealership
{
    public int Id { get; set; }

    public string Name { get; set; }
```

Листинг 26.2. Модель представления, используемая для создания отчета с показателями работы дилеров

```
public class DealershipPerformanceReport
{
    public DateTime ReportStartDate { get; set; }

    public DateTime ReportEndDate { get; set; }

    public List<DealershipPerformanceStatus> Dealerships { get; set; }
}

public class DealershipPerformanceStatus
{
    public string DealershipName { get; set; }

    public int TotalRevenue { get; set; }

    public int TargetRevenue { get; set; }

    public int NetProfit { get; set; }

    public int TargetProfit { get; set; }
}
```

Листинг 26.3. Отображение предметных объектов в модель представления

```
var statuses = new List<DealershipPerformanceStatus>();
foreach (var id in dealershipIds)
{
    // проблема N+1 запросов - источник низкой производительности и эффективности
    // повторное использование предметного кода - высокая скорость разработки
    var dealership = repository.Get(id);
    var targets = provider.Get(dealership, start, end);
    var actuals = calculator.CalculateFor(dealership, start, end);

    // отображение предметной модели в модель представления выполняется,
    // чтобы ослабить связь между пользовательским интерфейсом
    // и предметными объектами
    // эту логику можно перенести в отдельный компонент,
    // реализующий отображение
    statuses.Add(new DealershipPerformanceStatus
    {
        DealershipName = dealership.Name,
        TotalRevenue = actuals.TotalRevenue,
        TargetRevenue = targets.TargetRevenue,
        NetProfit = actuals.NetProfit,
        TargetProfit = targets.TargetProfit
    });
}

// логику отображения можно вынести в отдельный компонент,
// реализующий отображение
var viewModel = new DealershipPerformanceReport
{
}
```

```

ReportStartDate = start,
ReportEndDate = end,
Dealerships = statuses
};

```

ПРИМЕЧАНИЕ

Загрузить полный программный код примеров можно на странице <http://www.wrox.com/go/domaindrivendesign>, на вкладке Download Code (Загружаемый код).

Как можно судить по листингам с 26.1 по 26.3, создать отчет с использованием имеющихся предметных объектов не представляет никакой сложности. Одно из важнейших решений, которые придется принять, — где разместить логику отображения. Ее можно оформить в виде прикладной службы или контроллера. Вы можете создать отдельные классы, осуществляющие отображение, или реализовать это отображение внутри модели представления — в конструкторе или в статическом фабричном методе. В листинге 26.4 показано, как может выглядеть решение, основанное на создании прикладной службы с именем `DealershipReportBuilder`.

Листинг 26.4. Прикладная служба, которая создает отчет и выполняет все необходимые отображения

```

public class DealershipPerformanceReportBuilder
{
    private IDealershipRepository repository;
    private IDealershipRevenueCalculator calculator;
    private IDealershipPerformanceTargetsProvider provider;

    public DealershipPerformanceReportBuilder(IDealershipRepository repository,
                                              IDealershipRevenueCalculator calculator,
                                              IDealershipPerformanceTargetsProvider provider)
    {
        this.repository = repository;
        this.calculator = calculator;
        this.provider = provider;
    }

    public DealershipPerformanceReport BuildReport(IEnumerable<int> dealershipIds,
                                                  DateTime start, DateTime end)
    {
        var statuses = BuildStatuses(dealershipIds, start, end);

        return new DealershipPerformanceReport
        {
            ReportStartDate = start,
            ReportEndDate = end,
            Dealerships = statuses
        };
    }

    private List<DealershipPerformanceStatus> BuildStatuses(
        IEnumerable<int> dealershipIds, DateTime start, DateTime end)

```

```
{  
    var statuses = new List<DealershipPerformanceStatus>();  
    foreach (var id in dealershipIds)  
    {  
        var dealership = repository.Get(id);  
        var targets = provider.Get(dealership, start, end);  
        var actuals = calculator.CalculateFor(dealership, start, end);  
  
        statuses.Add(new DealershipPerformanceStatus  
        {  
            DealershipName = dealership.Name,  
            TotalRevenue = actuals.TotalRevenue,  
            TargetRevenue = targets.TargetRevenue,  
            NetProfit = actuals.NetProfit,  
            TargetProfit = targets.TargetProfit  
        });  
    }  
  
    return statuses;  
}  
}
```

К сожалению, выгоды не даются бесплатно. Повторное использование предметных объектов из листинга 26.2 позволяет легко справиться с задачей создания отчета. Однако из-за этого логика создания модели представления в листинге 26.4 может иметь крайне низкую производительность. Для каждого идентификатора дилера извлекается информация о соответствующем дилере и показателях его работы. С этой целью инструментом объектно-реляционного отображения может быть сгенерировано до трех запросов к базе данных. В системе, где имеется десять дилеров, может быть выполнено тридцать запросов, которые могут приводить к серьезным последствиям при пиковых нагрузках. Производительность не всегда является важным фактором, но в сценариях создания отчетов, подобных рассматриваемому, важно знать, сколько запросов выполняется и как они могут помешать нормальному функционированию, особенно если в работу вовлечены инструменты ORM с поддержкой отложенной загрузки.

Еще одна проблема, связанная с использованием отображений, заключается в возможной необходимости раскрыть дополнительные свойства предметных объектов, которые предпочтительнее было бы оставить скрытыми внутри предметной модели. Как следствие, это приведет к более тесной зависимости уровня прикладных служб от структуры предметной модели. Чтобы избавиться от нежелательной зависимости, можно прибегнуть к другим шаблонам проектирования, таким как «Посредник» (mediator).

С использованием шаблона «Посредник»

Чтобы создать модель представления с информацией, необходимой для отчета, но не имеющую тесной зависимости от структуры предметной модели, можно использовать шаблон проектирования «Посредник» (mediator). В соответствии с этим шаблоном модель представления передается посреднику, который затем

передается в предметную модель. Предметные объекты взаимодействуют с посредником, который обновляет модель представления. Это не приводит к нарушению границ уровней, потому что посредник реализует интерфейс, принадлежащий предметной модели.

Как часть альтернативной реализации отчета с показателями работы дилеров, в листинге 26.5 демонстрируется интерфейс посредника вместе с измененными версиями предметных объектов, которые больше не экспортируют свою структуру, но предоставляют методы для взаимодействия с посредником.

Листинг 26.5. Интерфейс посредника и предметные объекты, использующие его

```
// интерфейс посредника - постоянная структура предметной модели, которая
// может быть доступна уровню прикладных служб
public interface IDealershipAssessment
{
    int TotalRevenue { get; set; }

    int TargetRevenue { get; set; }

    int NetProfit { get; set; }

    int TargetProfit { get; set; }
}

public class DealershipPerformanceTargets
{
    // приватные поля скрывают потенциально изменяемую часть предметной модели
    private int targetRevenue;
    private int targetProfit;

    public void Populate(IDealershipAssessment mediator)
    {
        mediator.TargetRevenue = targetRevenue;
        mediator.TargetProfit = targetProfit;
    }
}

public class DealershipPerformanceActuals
{
    // приватные поля скрывают потенциально изменяемую часть предметной модели
    private int totalRevenue;
    private int netProfit;

    public void Populate(IDealershipAssessment mediator)
    {
        mediator.TotalRevenue = totalRevenue;
        mediator.NetProfit = netProfit;
    }
}
```

IDealershipAssessment в листинге 26.5 — это интерфейс посредника. Всякий раз, когда конкретная реализация посредника передается в вызов метода Populate()

класса `DealershipPerformanceTargets` или `DealershipPerformanceActuals`, в поля объекта-посредника записываются значения частных переменных экземпляра. Преимущество такого решения заключается в отсутствии необходимости экспортировать частные переменные за пределы предметной модели. В отсутствие тесных связей они могут изменяться как угодно. Это существенно отличает данный пример от предыдущего. Для большей ясности в листинге 26.6 приводится реализация посредника.

Листинг 26.6. Реализация посредника

```
// Окончание "mediator" в имени класса использовано исключительно
// для демонстрационных целей
public class DealershipAssessmentMediator : IDalershipAssessment
{
    private DealershipPerformanceStatus status;
    public DealershipAssessmentMediator(DealershipPerformanceStatus status)
    {
        this.status = status;
    }

    public int TotalRevenue
    {
        get { return status.TotalRevenue; }
        set { status.TotalRevenue = value; }
    }

    public int TargetRevenue
    {
        get { return status.TargetRevenue; }
        set { status.TargetRevenue = value; }
    }

    public int NetProfit
    {
        get { return status.NetProfit; }
        set { status.NetProfit = value; }
    }

    public int TargetProfit
    {
        get { return status.TargetProfit; }
        set { status.TargetProfit = value; }
    }
}
```

Объект посредника `DealershipAssessmentMediator`, представленный в листинге 26.6, обортывает модель представления `DealershipPerformanceStatus`. Когда посредник передается предметным объектам, они устанавливают соответствующие свойства посредника. В свою очередь, посредник устанавливает свойства модели представления `DealershipPerformanceStatus`, заключенной в нем. Благодаря

такой организации предметная модель и модель представления остаются независимыми друг от друга, как при обычном отображении.

Решение о выборе шаблона «Посредник» во многом основывается на опыте, суждениях и нескольких ключевых критериях. Если есть необходимость передавать наружу приватное состояние предметной модели, шаблон «Посредник» должен находиться в числе первых рассматриваемых альтернатив. Однако если с расширением предметной модели нежелательно увеличивать сложность посредника, выбор этого шаблона, вероятно, будет не самым оптимальным. Отчеты с высокой производительностью — еще один сценарий, в котором может быть желательно отказаться от шаблона «Посредник» из-за недостаточной управляемости. Если для вас производительность является важным фактором, рассмотрите возможность прямого обращения к хранилищу данных.

Прямое обращение к хранилищу данных

В тех случаях, когда производительность и эффективность играют важную роль или отсутствует желание «продираться» через наслоения сложностей и уровни отображения, многие практики DDD извлекают информацию для своих отчетов прямо из базы данных. В приложениях, реализующих шаблон CQRS, для каждого отчета создаются специализированные, денормализованные копии данных. В приложениях, не использующих шаблон CQRS, для выполнения запросов к базам данных обычно используются низкоуровневые технологии, такие как ADO.NET. Однако нередко используются и такие низкоуровневые возможности инструментов ORM, как механизм запросов HQL во фреймворке NHibernate.

ПРИМЕЧАНИЕ

Шаблон CQRS подробно рассматривался в главе 24 «CQRS: архитектура ограниченного контекста».

В этом разделе будут представлены примеры обращений к главному хранилищу данных проекта с использованием запросов, специально предназначенных для составления отчетов. Затем будет показан пример получения денормализованной копии данных (кэшированного представления), используемой специально для отчетов. Каждый из примеров демонстрирует создание отчета лояльности для электронного магазина спорттоваров. Этот отчет помогает определить успешность программы лояльности. Формат отчета приводится в табл. 26.1.

Таблица 26.1. Формат отчета лояльности

	Кол-во очков (на \$)	Чистая прибыль (процент от общей суммы)	Кол-во регистраций	Кол-во покупок (процент от общего числа)
Месяц А
Месяц Б

Объем прибыли, который генерирует программа лояльности, является самым важным показателем в отчете. Как можно видеть в табл. 26.1, этот параметр представлен процентом от общей прибыли, полученным по программе лояльности для того или иного месяца. Для оптимизации стратегии лояльности и успешного соперничества с конкурирующими компаниями электронный магазин спортивных товаров часто определяет число премиальных очков. С помощью отчета лояльности руководство предприятия может сделать вывод о том, как влияет это соотношение на общий успех программы. Наконец, никакой отчет нельзя признать полным без информации о популярности ресурса, поэтому в отчет лояльности включено также число зарегистрированных покупателей.

Обращение к хранилищу

Создание отчетов прямым обращением к хранилищу данных дает более полный контроль и позволяет писать эффективные запросы. Чтобы сгенерировать отчет лояльности, как показано в табл. 26.1, в SQL-запросе может понадобиться выполнить соединение множества таблиц, включая `orders`, `users`, `loyaltyAccounts`, `loyaltySettings` и, возможно, каких-либо других. Многие разработчики считают, что доверять инструментам ORM выполнение сложных запросов с соединением множества таблиц, как в данном примере, — это верный путь к неудаче. Как результат, все большую популярность стали завоевывать микрофреймворки ORM, потому что они дают некоторые выгоды, как и большие фреймворки, но выключаются из игры в особенно сложных случаях. Микрофреймворки ORM находятся на более низком уровне, нежели большие фреймворки, предоставляя больший контроль над запросами и упрощая возможность их оптимизации.

В листинге 26.7 демонстрируется прикладная служба, использующая `Dapper` (<https://code.google.com/p/dapper-dot-net/>), компактный микрофреймворк ORM, которая выполняет SQL-запросы непосредственно к основной базе данных SQL проекта, не вовлекая в работу предметную модель. В листинге 26.8 приводятся определения моделей базы данных и представления, между которыми происходит отображение.

ВНИМАНИЕ

SQL-запросы, представленные в этой главе, не рекомендуется использовать как основу. Их цель лишь в том, чтобы продемонстрировать возможность более полного контроля, когда требуется организовать быстрый доступ к данным.

Листинг 26.7. Прямой доступ к базе данных SQL с помощью `Dapper`

```
public LoyaltyReport Build(DateTime start, DateTime end)
{
    IEnumerable<PurchasesAndProfit> profits;
    IEnumerable<SignupCount> signups;
    IEnumerable<LoyaltySettings> settings;

    using(var con = new SqlConnection(connString))
    {
```

```

con.Open();

var pointsQuery =
    "select [Month], [PointsPerDollar] from loyaltySettings "
    + "where [Month] >= @start "
    + "and [Month] >= @end";
settings = con.Query<LoyaltySettings>(pointsQuery,
                                     new {start=start, end=end});

var signupsQuery = "select count(*) from loyaltyAccounts" +
    "where isActive = true " +
    "where [created] >= @start " +
    "and [created] < @end ";
signups = con.Query<SignupCount>(signupsQuery,
                                new {start=start, end=end } );

var profitQuery =
    "select " +
    "concat(month(o.[date]), '/', year(o.[date])) as Month, " +
    "(select ((cast(count(*) as decimal) / (" +
    "    select count(*) from orders" +
    "    where [date] >= @start" +
    "    and [date] < @end" +
    ")) * 100)) as Purchases," +
    "(select ((sum(netProfit) / (" +
    "    select sum(netProfit) from orders" +
    "    where [date] >= @start" +
    "    and [date] < @end " +
    ")) * 100)) as NetProfit" +
    "from orders o" +
    "join Users u on o.userId = u.id" +
    "join LoyaltyAccounts la on u.id = la.userId" +
    "where la.isActive = 1" +
    "and o.[date] >= @start" +
    "and o.[date] < @end" +
    "group by concat(month(o.[date]), '/', year(o.[date]))";
profits = con.Query<PurchasesAndProfit>(profitQuery,
                                       new {start=start, end=end});
}

return Map(profits, signups, settings, start, end);
}

```

Листинг 26.8. Модели представления и базы данных для отчета лояльности

```

// Модели представления
public class LoyaltyReport
{
    public IEnumerable<LoyaltySummary> Summaries { get; set; }
}

public class LoyaltySummary
{

```

```
public DateTime Month { get; set; }

public int PointsRatio { get; set; }

public double NetProfit { get; set; }

public int SignUps { get; set; }

public int Purchases { get; set; }
}

// Модели базы данных
public class LoyaltySettings
{
    public DateTime Month { get; set; }

    public int PointsPerDollar { get; set; }
}

public class SignupCount
{
    public DateTime Month { get; set; }

    public int Signups { get; set; }
}

public class PurchasesAndProfit
{
    public DateTime Month { get; set; }

    public int Purchases { get; set; }

    public double Profit { get; set; }
}
```

Как можно заметить в листинге 26.7, микрофреймворк Dapper добавляет в класс `SqlConnection` из ADO.NET метод расширения `Query<T>`. Этот метод отображает результаты запроса, переданного в аргументе, в объект типа `T`, который он создает автоматически. Но самое важное в листинге 26.7 то, что разработчик полностью контролирует генерируемый SQL-запрос. В сценариях, где важна высокая производительность, низкоуровневый доступ к данным помогает обойти неэффективность, свойственную фреймворкам ORM. К сожалению, более полный контроль часто влечет за собой дублирование кода.

ПРИМЕЧАНИЕ

Полный код этого и других примеров для данной главы можно найти в загружаемых файлах, включая реализацию `Map()` и проект базы данных SQL Server с представленной выше схемой. Попробуйте поэкспериментировать с фреймворками ORM и сравнить эффективность SQL-запросов, генерируемых ими, на больших наборах данных.

ПРИМЕЧАНИЕ

Микрофреймворки ORM — это библиотеки удобных вспомогательных методов, иногда даже включающих поддержку предметно-ориентированного языка (Domain-Specific Language, DSL), для выполнения запросов к базам данных. Приставка «микро» в названии означает, что они не пытаются спрятать базу данных за слоем абстракций; они всего лишь немного упрощают выполнение запросов. Ничто не вынуждает вас использовать микрофреймворки ORM; вы можете пользоваться ADO.NET или эквивалентными низкоуровневыми библиотеками для работы с любыми хранилищами.

Одна из проблем приема прямого доступа к хранилищу — дублирование кода. Некоторые предметные сущности имеют вычисляемые свойства. Если взглянуть на SQL-код для `profitQuery` в листинге 26.7, можно заметить, что процент чистой прибыли от программы лояльности вычисляется с учетом общей прибыли за тот же период. Весьма вероятно, что эти вычисления производятся также где-то в предметной модели. В результате возникает риск нарушения принципа «не повторяться» (DRY), и, как следствие, при изменении логики вычислений по каким-либо причинам придется изменить SQL-код запроса в отчете и предметную логику, что порой легко упустить из виду или забыть.

Дублирование предметной логики в каком-либо еще месте не лучший выход. Понятно, что если изменить предметную логику и забыть изменить логику запроса, можно получить проблемы, раздражающие пользователей, или отчет с полностью неверными числами. Если вас волнует эта проблема, подумайте о возможности сохранения значений вычисляемых свойств. В этом случае потребуется вычислять значение свойства и сохранять его в базе данных при каждом обновлении. Однако если обновления требуется вносить в нескольких местах в базе данных, значение нужно будет повторно вычислять во множестве мест или добавлять в базу данных триггер во избежание этого.

ПРИМЕЧАНИЕ

По общему мнению, следует избегать включения любой бизнес-логики в базы данных, потому что это усложняет чтение, сопровождение и тестирование кода. Иногда, впрочем, без этого не обойтись, однако важно понимать все «за» и «против», чтобы можно было принять обоснованное решение.

Чтение денормализованных кэшированных представлений

Иногда даже прямое обращение к базе данных с использованием SQL-запросов, написанных вручную, может оказаться неэффективным. По этой причине некоторые практики DDD создают денормализованные копии данных (кэшированные представления). Концептуально этот подход близок к шаблону CQRS. Всякий раз, когда происходят изменения, изменяется основная база данных, а также соответствующие денормализованные кэшированные представления. Рисунок 26.1 демонстрирует, как можно реализовать этот шаблон для отчета лояльности.

Когда размещается новый заказ и регистрируется новый пользователь (проявляется как вызовы методов, команды, предметные события и т. д.), вызов пред-

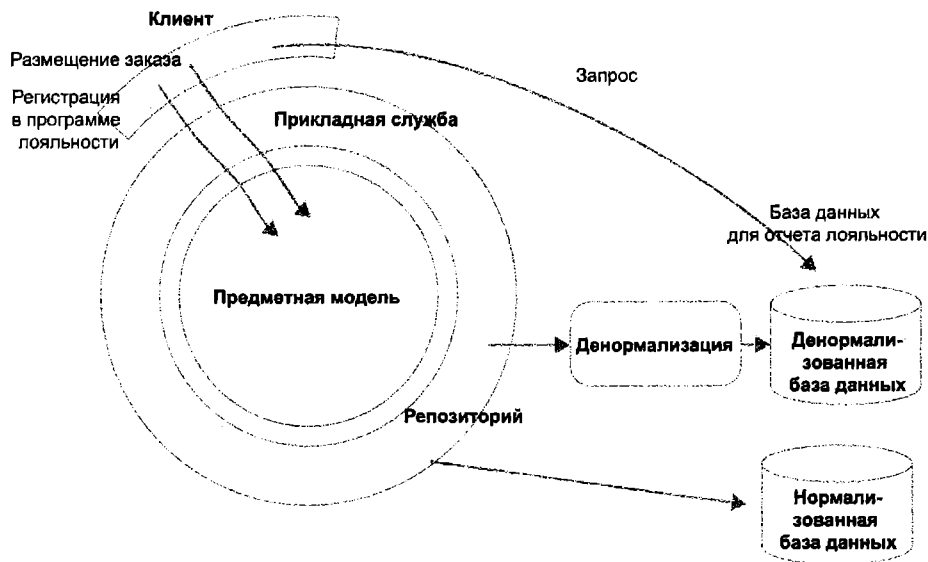


Рис. 26.1. Денормализованное кэшированное представление для отчета лояльности

метной модели происходит как обычно. Однако когда создаются денормализованные представления, обновления обычно следуют по одному пути в основную базу данных и по меньшей мере еще по одному пути — в денормализованное кэшированное представление через средство денормализации, как показано на рис. 26.1. Задача средства денормализации обычно заключается в том, чтобы реструктурировать данные для упрощения их извлечения с применением простых инструкций SQL. В листинге 26.9 показана альтернативная реализация `LoyaltyReportBuilder`, извлекающая данные из денормализованного представления и подчеркивающая, насколько простым может быть запрос, если все сложные преобразования возложить на средство денормализации.

Листинг 26.9. Простота извлечения данных при использовании денормализованного кэшированного представления

```

public LoyaltyReport Build(DateTime start, DateTime end)
{
    IEnumerable<LoyaltySummary> summaries;

    using(var con = new SqlConnection(connString))
    {
        con.Open();
        var query =
            "select [Month], PointsPerDollar, NetProfit, Signups, Purchases " +
            "from denormalizedLoyaltyReportViewCache " +
            "where [Month] >= @start " +
            "and [Month] < @end";
        summaries = con.Query<LoyaltySummary>(query,

```

```

        new {start = start, end = end});
    }

    return new LoyaltyReport
    {
        Summaries = summaries
    };
}

```

Как можно видеть в листинге 26.9, сложность запроса уменьшилась до единственной простой SQL-инструкции `select` без соединений таблиц. Все это — благодаря предварительным усилиям по денормализации данных. Прежде чем использовать этот подход, вы должны решить, дает ли он достаточное уменьшение сложности или увеличение производительности. Впрочем, вы можете свободно комбинировать разные подходы в своих проектах, если это представляется возможным.

Создание проекций из потоков событий

Приложения, использующие приемы регистрации событий, описанные в главе 22 «Регистрация событий», требуют других подходов к созданию отчетов, потому что они не хранят представления своего текущего состояния. Вместо этого приложения с поддержкой регистрации событий полагаются на проекции. *Проекции* — это в действительности самые обычные запросы к потокам событий, которые возвращают желаемое представление состояния или новые потоки, основанные на содержимом событий в оригинальном потоке.

Использование проекций в контексте создания отчетов будет продемонстрировано в следующих примерах, где проекции используются для создания отчета диагностики заболеваний. Сотрудники отдела здравоохранения используют этот отчет для отслеживания количества диагнозов разных заболеваний по месяцам. Формат этого отчета приводится в табл. 26.2.

Таблица 26.2. Формат отчета диагностики заболеваний

	02/2014		03/2014		04/2014		05/2014	
	ВСЕГО	%	ВСЕГО	%	ВСЕГО	%	ВСЕГО	%
Диагноз А	—	—	—	—	—	—	—	—
Диагноз Б	—	—	—	—	—	—	—	—

Как видно в табл. 26.2, каждая строка в отчете диагностики заболеваний сообщает число диагнозов, поставленных в каждый месяц. Для каждого месяца отображается число поставленных диагнозов, а также процент от всех диагнозов, поставленных в этот месяц. Используя этот отчет, сотрудники отдела здравоохранения могут увидеть тенденции в определенных диагнозах. Это может помочь им понять сезонные изменения или увидеть связь с другими событиями, такими как появление новой вакцины или введение новых приемов оздоровления.

Чтобы реализовать этот отчет, для каждого диагноза создается новый поток событий с именем в формате *диагноз-{идентификатор_диагноза}-{месяц}*. Эти новые потоки создаются из проекций, основанных на одном и том же потоке событий со всеми диагнозами (поток «диагнозы»). Этот процесс иллюстрирует рис. 26.2.

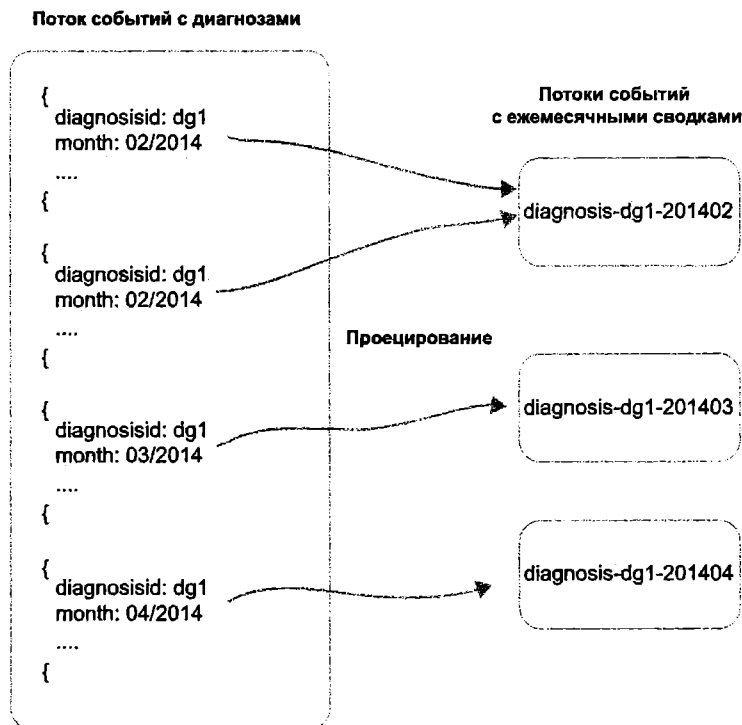


Рис. 26.2. Проекция общего потока событий «диагнозы» в потоки событий с ежемесячными сводками о диагностике заболеваний

Для каждого диагноза поток содержит все события его постановки за каждый месяц, как показано на рис. 26.2. Например, все диагнозы с идентификатором `dg1`, поставленные в феврале 2014 года, проецируются в поток `diagnosis-dg1_201402`. Таким образом, когда потребуется вывести отчет, достаточно будет лишь подсчитать число событий в этом месяце. Как видите, использование проекций напоминает философию создания денормализованных эшечированных представлений — вся основная работа выполняется до создания отчета в целях уменьшения сложности, связанной с чтением данных.

Настройка потоков событий для проецирования

Чтобы опробовать примеры из этого раздела, вам понадобится версия Event Store не ниже 3.0.0 rc2, в которой впервые появилась поддержка проекций (<https://geteventstore.com/downloads/>). Загрузив zip-файл, распакуйте его в папку по свое-

му выбору. Чтобы запустить Event Store, откройте консоль PowerShell с привилегиями администратора, перейдите в папку, куда был распакован архив, и выполните команду:

```
.\EventStore.SingleNode.exe --db .\ESData --run-projections=all
```

После запуска Event Store для включения опции поддержки проекций необходимо выполнить несколько настроек. Откройте в браузере веб-интерфейс (<http://localhost:2113/projections>), перейдите на вкладку Projections (Проекции) и запустите следующие проекции: `$by_category` и `$stream_by_category`.

ПРИМЕЧАНИЕ

Желающие опробовать описываемые здесь примеры могут воспользоваться утилитой в загружаемом коде, которая создает поток «diagnoses» и вставляет в него тестовые данные. Для этого нужно запустить веб-приложение PPPDDD.Reporting, перейти по адресу http://localhost:{номер_порта}/HealthcareEventProjectionReport. Если вы хотите увидеть, как она работает, или изменить тестовые данные, вы можете обратиться к исходному коду в классе HealthcareEventProjectionReportController.

ПРИМЕЧАНИЕ

Проект Event Store постоянно развивается. В какой-то момент в версиях выше v3 rc2 поддержка проекций была включена по умолчанию. Желающие могут задать интересные их вопросы в Google-группе Event Store (<https://groups.google.com/forum/#!forum/event-store>) или на дискуссионном форуме этой книги.

Создание проекций для отчетов

Проекции создаются на языке JavaScript и могут передаваться прикладному программному интерфейсу по протоколу HTTP, а также вводиться вручную в веб-интерфейсе администратора. В данном примере используется второй способ, для выполнения которого нужно сначала перейти на вкладку Projections (Проекции), затем выбрать New Projection (Создать проекцию). Так как проекция должна группировать все события постановки диагнозов за месяц, ей присвоено имя DiagnosesByMonth. Реализация этой проекции показана в листинге 26.10 и должна быть вставлена в окно редактора Source (Исходный код). В процессе создания проекции DiagnosesByMonth выберите режим Continuous (Непрерывный) и установите флажок Emit Enabled (Разрешить вывод). Закончив, щелкните на кнопке Post (Отправить), чтобы создать проекцию.

Листинг 26.10. Проекция на JavaScript, разбивающая поток диагнозов по месяцам

```
fromStream('diagnoses')
.whenAny(function(state, ev) {
  var date = ev.data.Date.replace('/', '');
  var diagnosisId = ev.data.DiagnosisId;
  linkTo('diagnosis-' + diagnosisId + '_' + date, ev);
});
```

Хранилище событий Event Store применяет проекции к каждому событию в потоке. Таким образом, код на JavaScript в листинге 26.10 применяется к каждому событию в потоке диагнозов. Для каждого из этих событий создается ссылка в другом потоке. Этот другой поток представляет все диагнозы с одинаковым идентификатором `diagnosisId`, поставленные в одном месяце. Это тот самый процесс, что изображен на рис. 26.2. Event Store поддерживает такое поведение проекций с помощью метода `linkTo()`, который добавляет ссылку на указанное событие, переданную во втором аргументе, в поток, указанный в первом аргументе, создавая его при необходимости. То есть проекции не копируют сами события; они лишь создают ссылки, или указатели.

Чтобы убедиться, что проекция действует, перейдите на вкладку **Streams** (Потоки) в веб-интерфейсе Event Store и понаблюдайте за именами вновь созданных потоков событий. Вы должны увидеть потоки событий с именами в формате `diagnosis-{diagnosisId}_{month}`, такие как `diagnosis-d13_201402`. Если щелкнуть на одном из таких потоков, можно увидеть указатели на события в потоке `diagnoses`, включенные в проекцию.

Подсчет числа событий в потоке

В каждой строке отчета должно отображаться число диагнозов, поставленных за месяц. Как обсуждалось ранее, эти числа просто соответствуют количеству событий в каждом потоке, созданном проекцией из листинга 26.10. Один из способов определить размер потока событий — создать еще одну проекцию. Этот подход рекомендуется многими и будет использоваться в обсуждаемом примере.

Чтобы создать проекцию, подсчитывающую число событий в месяце для каждого диагноза, можно воспользоваться программным кодом на JavaScript из листинга 26.11. Чтобы иметь возможность следовать за описанием примера, назовите эту проекцию `DiagnosesByMonthCounts`. Она также должна использовать режим `Continuous` (Непрерывный), но на этот раз оставьте сброшенным флажок `Emit Enabled` (Разрешить вывод). После этого останется лишь создать проекцию щелчком на кнопке **Post** (Отправить).

Листинг 26.11. Проекция для получения числа событий во всех потоках, принадлежащих к категории

```
fromCategory('diagnosis')
  .foreachStream()
  .when({
    $init : function(s,e) {return {count : 0}};
    "diagnosis" : function(s,e) { s.count += 1} // изменение происходит на месте
  });
```

В Event Store *категориями* считаются потоки, имена которых начинаются с одинаковых префиксов. *Префикс* — это часть строки с именем потока, предшествующая дефису. Таким образом, все потоки с именами, начинающимися с `diagnosis-` и созданными первой проекцией, принадлежат к категории `diagnosis`. Поддержка категорий лежит в основе функционирования проекции из листинга 26.11. Метод `foreachStream()` применяется к каждому потоку из категории, соответственно проекция в листинге 26.11 выполнит обход всех потоков в категории `diagnosis`

и подсчитает количество событий в ней. Это количество хранится в состоянии проекции. Убедиться в этом вы сможете, обратившись к состоянию проекции, если при этом не забудете указать в параметре `partition` имя потока, состояние которого должно быть возвращено. Например, запрос `http://localhost:2113/projection/DiagnosesByMonthCounts/state?partition=diagnosis-dg1_201402` вернет число всех событий в потоке в следующем формате:

```
{
  count: 1
}
```

Создание необходимого числа потоков

Как отмечалось в главе 22, создание потоков событий в Event Store является относительно незатратной операцией. Поэтому чтобы получить общее число диагнозов, поставленных в любой заданный месяц, можно использовать проекции, создающие дополнительные потоки. Порядок действий при этом следует тому же шаблону, что и использовался в двух последних примерах. Сначала нужно выделить события, принадлежащие заданным месяцам, как показано в листинге 26.12, и применить те же настройки, что и для проекции `DiagnosesByMonth`. Чтобы следовать за данным примером, дайте этой проекции имя `Months`.

Листинг 26.12. Проекция для выделения диагнозов по месяцу

```
fromStream('diagnoses')
  .whenAny(function(state, ev) {
    var date = ev.data.Date.replace('/', '');
    linkTo('month-' + date, ev);
  });
```

После запуска проекции `Months` останется только подсчитать сумму чисел в потоках, как это делает проекция `DiagnosesByMonthCounts`. Реализация этой проекции представлена в листинге 26.13. Она должна показаться вам знакомой. После запуска этой проекции мы будем иметь все потоки, необходимые для создания отчета.

Листинг 26.13. Проекция для подсчета общего числа диагнозов в каждом месяце

```
fromCategory('month')
  .foreachStream()
  .when({
    $init : function(s,e) {return {count : 0}},
    "diagnosis" : function(s,e) { s.count += 1 } // изменение происходит на месте
  });
```

Создание отчета из потоков и проекций

После того как будет получено множество потоков событий со всей необходимой информацией, создание отчета сведется к последовательности HTTP-вызовов (или обращений к клиентской библиотеке) и отображению объектов. Прикладная служба с именем `HealthcareReportBuilder` демонстрирует эту заключительную часть данного примера. В листинге 26.14 представлена начальная версия `HealthcareReportBuilder`, содержащая верхнеуровневую логику создания отчета.

Листинг 26.14. Верхнеуровневая логика создания отчета

```
public class HealthcareReportBuilder
{
    public HealthcareReport Build(DateTime start, DateTime end,
        IEnumerable<string> diagnosisIds)
    {
        // столбцы отчета
        var monthsInReport = GetMonthsInRange(start, end).ToList();
        var monthlyOverallTotals = FetchMonthlyTotalsFromES(monthsInReport);
        var queries = BuildQueriesFor(
            monthsInReport, diagnosisIds).ToList();
        var summaries = BuildMonthlySummariesFor(
            queries, monthlyOverallTotals).ToList();

        return new HealthcareReport
        {
            Start = start,
            End = end,
            Summaries = summaries
        };
    }
    ...
}
```

Чтобы создать отчет `HealthcareReport`, служба `HealthcareReportBuilder` производит вычисления для каждого месяца в указанном диапазоне. Сначала она получает из Event Store общее число диагнозов, вызывая метод `FetchMonthlyTotalsFromES()`, реализация которого приводится в листинге 26.15.

Листинг 26.15. Получение из Event Store общего числа диагнозов за месяц

```
private Dictionary<DateTime, int> FetchMonthlyTotalsFromES(
    IEnumerable<DateTime> months)
{
    // Не следует жестко "зашивать" адрес URL в программный код.
    // Используйте ресурс точки входа.
    var projectionStateUrl = "http://localhost:2113/projection/MonthsCounts/state";

    var totals = new Dictionary<DateTime, int>();
    foreach(var m in months)
    {
        var streamName = "month-" + m.ToString("yyyyMM");
        var url = projectionStateUrl + "?partition=" + streamName;
        var response = new WebClient().DownloadString(url);
        var count = Json.Decode<DiagnosisCount>(response);
        totals.Add(m, count == null ? 0 : count.Count);
    }

    return totals;
}
```

Чтобы получить общее число диагнозов в каждом месяце, код в листинге 26.15 конструирует URL для доступа к ресурсу `state` проекции `MonthsCounts`. Имя по-

тока со всеми диагнозами для данного месяца передается в параметре `partition`. В ответ Event Store API возвращает сумму в формате JSON. Получить ее можно, отобразив JSON-ответ в объект `DiagnosisCount`, который является объектом переноса данных (DTO) со структурой, соответствующей структуре JSON-ответа, представленной в листинге 26.16. Свойство `Count` объекта затем сохраняется как общее число для месяца. Если в течение месяца не было поставлено ни одного диагноза, значение в JSON-ответе вообще будет отсутствовать. В этом случае код запишет нулевое значение.

Листинг 26.16. Объект переноса данных `DiagnosisCount`, соответствующий формату ответа

```
public class DiagnosisCount
{
    public int Count { get; set; }
}
```

После получения общего числа диагнозов, поставленных в каждом месяце, служба `HealthcareReportBuilder` затем получает общее число каждого диагноза в месяце. Перед обращением к Event Store, однако, выполняется промежуточный шаг в виде вызова `BuildQueriesFor()`. Метод `BuildQueriesFor()` создает коллекцию строго типизированных объектов DTO, имеющих формат, как определено в листинге 26.17, для большей выразительности программного кода.

Листинг 26.17. Объект `DiagnosisQuery`

```
public class DiagnosisQuery
{
    public DateTime Month { get; set; }

    public string DiagnosisId { get; set; }
}
```

После создания коллекции объектов `DiagnosisQuery` служба `HealthcareReportBuilder` использует ее для выполнения заключительного запроса к Event Store вызовом `BuildMonthlySummariesFor()`, чтобы получить суммарную информацию о каждом диагнозе по месяцам. Своей реализацией он напоминает метод `FetchMonthlyTotalsFromES()` и выполняет всю фактическую работу, запрашивая Event Store и отображая ответ в объекты, как показано в листинге 26.18.

Листинг 26.18. Получение суммарной информации о каждом диагнозе по месяцам из Event Store

```
private IEnumerable<DiagnosisSummary>
BuildMonthlySummariesFor(IEnumerable<DiagnosisQuery> queries,
    Dictionary<DateTime, int> monthlyTotals)
{
    // для большей производительности запросы могут выполняться параллельно
    foreach (var q in queries)
    {
        var diagnosisTotal = FetchTotalFromESFor(q);
        var monthTotal = monthlyTotals[q.Month];
        var percent = monthTotal == 0 ? 0:
```

```

                                ((decimal)diagnosisTotal/monthTotal)*100;
yield return new DiagnosisSummary
{
    Amount = diagnosisTotal,
    DiagnosisName = GetDiagnosisName(q.DiagnosisId),
    Month = q.Month,
    Percentage = percent,
};
}
}

private int FetchTotalFromESFor(DiagnosisQuery query)
{
    // Не следует жестко "зашивать" адрес URL в программный код.
    // Используйте ресурс точки входа.
    var projectionStateUrl =
        "http://localhost:2113/projection/DiagnosesByMonthCounts/state";

    var streamname = "diagnosis-" + query.DiagnosisId + "_" +
        query.Month.ToString("yyyyMM");

    // здесь можно использовать кэширование
    var response = new WebClient().DownloadString(projectionStateUrl +
        "?partition=" + streamname);
    var count = Json.Decode<DiagnosisCount>(response);

    return count == null ? 0 : count.Count;
}

```

Кроме извлечения суммарной информации, `BuildMonthlySummaries()` вычисляет проценты, используя общее число диагнозов за месяц, полученное прежде, и отображает результаты в `DiagnosisSummary`. По завершении каждый объект `DiagnosisSummary` отображается в модель представления `HealthcareReport`, представленную в листинге 26.14. На этом самая сложная часть заканчивается, и остается лишь вывести отчет.

ВНИМАНИЕ

Как можно заметить в листинге 26.14, извлечение суммарной информации для каждого диагноза в каждый месяц может привести к большому числу запросов HTTP. Однако возвращаемые ими значения никогда не изменяются, поэтому можно использовать агрессивную стратегию кэширования для улучшения производительности.

ПРИМЕЧАНИЕ

Использование проекций для создания потоков может привести к появлению большого числа потоков внутри экземпляра или кластера Event Store. Показатели производительности и использования дискового пространства, несомненно, нужно контролировать, однако обычно нет необходимости уделять этому слишком пристальное внимание. База данных Event Store предусматривает возможность хранения миллионов потоков (<http://geteventstore.com/blog/20130210/the-cost-of-creating-a-stream/>).

Предметная отчетность по нескольким ограниченным контекстам

К сожалению, создание отчетов не всегда реализуется простым обращением к единственному хранилищу данных. В распределенных системах, подобных тем, что обсуждались во второй части книги «Стратегические шаблоны: взаимодействие ограниченных контекстов», каждый ограниченный контекст имеет собственное хранилище, что требует приложения дополнительных усилий для составления отчетов. В этом разделе будут представлены два подхода, опирающиеся на приемы, которые были рассмотрены в предыдущих главах. Один из них заключается в использовании принципов событийно-ориентированного программирования из главы 12 и создании выделенного ограниченного контекста для отчетов, который подписывается на множество событий и собирает всю информацию в локальной базе данных. Тем не менее в некоторых случаях можно использовать более легковесный подход, основанный на использовании приемов создания составного пользовательского интерфейса, описанных в главе 23 «Конструирование пользовательских интерфейсов приложения».

С использованием приемов составных пользовательских интерфейсов

Комбинирование данных из нескольких ограниченных контекстов для формирования отчета вполне реализуемо на практике, но обычно такой подход используется, только когда большую часть обработки можно выполнить в разных ограниченных контекстах. Любые дополнительные операции, такие как преобразование идентификаторов в имена, могут выполняться позднее, путем обращения к ограниченному контексту, владеющим искомой информацией. Для демонстрации этого можно использовать отчет сравнения территориальной популярности музыкальных альбомов. Сетевые компании, занимающиеся трансляцией музыки, могут использовать такой отчет для выяснения популярности каждого музыкального альбома в разных странах. Популярность определяется на основе общего числа трансляций и числа загрузок каждой песни, принадлежащей альбому. В табл. 26.3 показано, как выглядит такой отчет.

Таблица 26.3. Отчет сравнения популярности музыкальных альбомов на разных территориях

	Северная Америка	Европа	Азия
Альбом 1	–	–	–
Альбом 2	–	–	–
Альбом 3	–	–	–

Одна из самых больших проблем, возникающих при составлении отчета сравнения популярности музыкальных альбомов, состоит в том, что трансляции и загрузки — полностью независимые разделы бизнеса, со своими ограниченными контекстами. Поэтому чтобы получить общее число трансляций и загрузок для



Рис. 26.3. Объединение данных из разных ограниченных контекстов в один общий отчет

каждого альбома, нужно получить информацию из каждого ограниченного контекста, как показано на рис. 26.3.

К счастью, объединение можно реализовать в виде отдельного этапа. Общее число загрузок извлекается из ограниченного контекста загрузок, а общее число трансляций каждого альбома — из ограниченного контекста трансляций. Применение агрегирования на стороне клиента или на стороне сервера, как было продемонстрировано в главе 23, позволит с легкостью получить суммарную информацию для каждого музыкального альбома и для каждой территории.

Отдельный контекст отчетов

Производительность, эффективность и удобство, которые дает хранение в одном месте всех данных, необходимых для отчетов, иногда оказывается важным критерием. Такое объединенное хранилище может понадобиться, например, когда заинтересованные лица пожелают иметь возможность всестороннего анализа данных различными способами, для реализации которой предприятия часто нанимают специалистов по обработке и анализу данных. После прочтения второй части «Стратегические шаблоны: взаимодействия между ограниченными контекстами», где рассказывается о создании распределенных ограниченных контекстов, вы уже знаете, что по умолчанию это невозможно, так как каждый ограниченный контекст имеет собственное хранилище и слабо связан с другими контекстами. Но вам также известно, что взаимодействия с ограниченными контекстами посредством событий открывают возможность создания специального контекста отчетов, который мог бы подписаться на события от множества других ограниченных контекстов и собирать все необходимые ему данные.

Реализации контекстов отчетов в разных областях и системах могут иметь кардинальные различия. В простейших случаях они могут походить на любые другие ограниченные контексты, подписываясь на события и сохраняя их в базе данных SQL, как показано на рис. 26.4. С другой стороны, они могут извлекать информацию с применением самых разных технологий хранения данных, механизмов выработки рекомендаций и алгоритмов машинного обучения, таких как Netflix (<http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html>), как показано на рис. 26.5.

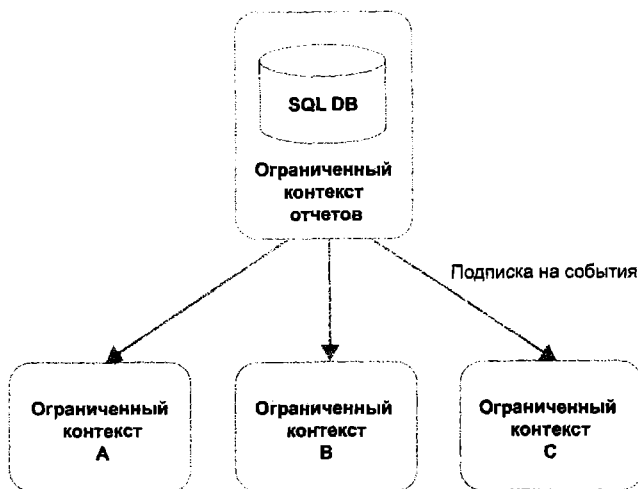


Рис. 26.4. Обычный контекст отчетов

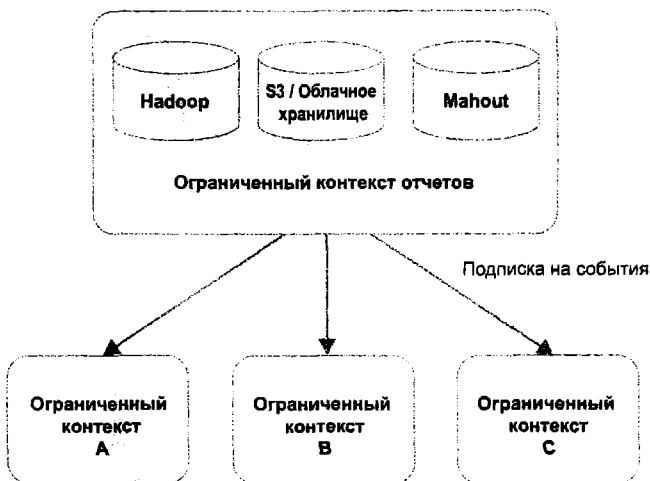
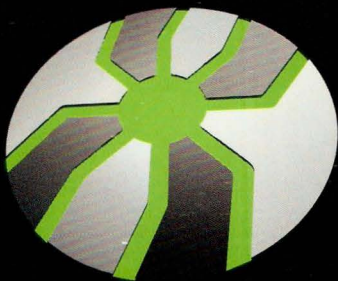


Рис. 26.5. Контекст отчетов со сложной обработкой данных

Дополнительную информацию об отчетах и анализе деловых данных в событийно-ориентированных системах с архитектурой SOA можно найти в статье Арнона Ротем-Гал-Оза (Arnon Rotem-Gal-Oz) на сайте InfoQ (<http://www.infoq.com/articles/BI-and-SOA>).

Ключевые идеи

- Отчеты можно создавать с помощью разных инструментов и технологий, позволяющих избегать образования тесных связей с предметной моделью.
- Некоторые отчеты оперируют данными, которые находятся внутри одного ограниченного контекста, тогда как другие могут потребовать извлекать информацию из множества ограниченных контекстов.
- Отображение предметных объектов в модели представления часто является самым быстрым решением, которое тем не менее не дает возможности управлять низкоуровневым доступом к данным.
- Для создания отчетов и ослабления зависимостей можно использовать такие шаблоны проектирования, как «Посредник» (mediator).
- Для обеспечения высокой эффективности запросов вполне допустимо напрямую обращаться к хранилищу данных, но стоит помнить о дублировании понятий и нарушении принципа «не повторяйся» (DRY).
- Обращение к основной базе данных, а также к денормализованным кэшированным представлениям — это два способа прямого доступа к данным.
- Использование денормализованных кэшированных представлений позволяет перенести всю основную работу в процесс денормализации и получить возможность использовать простые запросы для извлечения данных.
- Использование проекций с потоками событий также позволяет упростить чтение за счет обработки в фоновом режиме.
- Запрашивать данные из множества ограниченных контекстов в одних случаях можно с применением приемов, используемых в составных пользовательских интерфейсах, а в других — за счет отдельного контекста отчетов.



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

АНТИВИРУСНЫЕ
ПРОГРАММНЫЕ ПРОДУКТЫ

Писать программы легко — во всяком случае, с нуля. Но изменить однажды написанный программный код, который создали другие разработчики или вы сами каких-то шесть лет тому назад, — гораздо сложнее. Программа работает, но вы не знаете точно, как именно. Даже обращение к экспертам в предметной области ничего не дает, поскольку в коде не сохранилось никаких следов привычного для них языка.



Предметно-ориентированное проектирование (Domain-Driven Design, DDD) — это процесс тесной увязки программного кода с реалиями предметной области. Благодаря ему добавление в программный продукт новых возможностей по мере его развития становится таким же простым, как и при создании программы с нуля.

Эта книга в полной мере соответствует философии DDD и позволяет разработчикам перейти от философских рассуждений к решению практических задач. Она делится на четыре части: часть I посвящена философии, принципам и приемам предметно-ориентированного проектирования; в части II подробно обсуждаются стратегические шаблоны интеграции ограниченных контекстов; часть III охватывает тактические шаблоны создания эффективных моделей предметной области; часть IV в деталях описывает шаблоны проектирования, которые позволяют извлекать пользу из модели предметной области и создавать эффективные приложения.

 **ПИТЕР®**

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин

 vk.com/piterbooks
 instagram.com/piterbooks
 facebook.com/piterbooks
 youtube.com/ThePiterBooks

 **wrox™**
A Wiley Brand

