

ФРЕДЕРИК БРУКС МЛАДШИЙ

ОЧЕРКИ О ПРОГРАММНОЙ ИНЖЕНЕРИИ



МИФИЧЕСКИЙ ЧЕЛОВЕКО- МЕСЯЦ

ИЛИ КАК СОЗДАЮТСЯ ПРОГРАММНЫЕ СИСТЕМЫ



The Mythical Man-Month

*Essays on Software Engineering
Anniversary Edition*

Frederick P. Brooks, Jr.

University of North Carolina at Chapel Hill

ФРЕДЕРИК БРУКС
МЛАДШИЙ

МИФИЧЕСКИЙ ЧЕЛОВЕКО- МЕСЯЦ

ИЛИ КАК СОЗДАЮТСЯ
ПРОГРАММНЫЕ СИСТЕМЫ



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

2021

ББК 32.973.2-018
УДК 004.4
Б89

Брукс Фредерик

Б89 Мифический человеко-месяц, или Как создаются программные системы. — СПб.: Питер, 2021. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1636-2

Эта книга — юбилейное (дополненное и исправленное) издание — стала своего рода библией для разработчиков программного обеспечения во всем мире. Первое издание этой книги было написано Бруксом еще в 1975 году. И с тех пор считается, что каждый руководитель программного проекта должен прочитать этот труд. Прошло много лет, но актуальность написанного не уменьшается, хотя технологии и продвинулись далеко вперед. Ведь проекты продолжают проваливаться из-за недостатка времени, привлечение дополнительных сотрудников на конечных стадиях работы замедляет процесс, а формула минимального времени выполнения продолжает действовать.

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

ББК 32.973.2-018
УДК 004.4

На обложке использована иллюстрация А. Tobin «Dinosaurs». Scientific American, 51(22): p. 343. November 29, 1884.

Права на издание получены по соглашению с Frederick P. Brooks Jr. 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. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

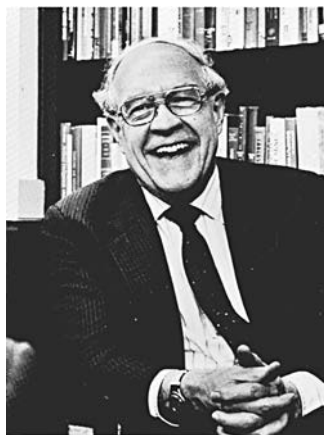
ISBN 978-0201835953 англ.
ISBN 978-5-4461-1636-2

© 1975, 1995 Frederick P. Brooks, Jr.
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Библиотека программиста», 2021

ОГЛАВЛЕНИЕ

Об авторе	7
Предисловие научного редактора к русскому изданию 2020 года	8
Предисловие к изданию 1995 года	19
Предисловие к первому изданию	22
Глава 1. Смоляные ямы	27
Глава 2. Мифический человеко-месяц	37
Глава 3. Хирургическая бригада	53
Глава 4. Аристократия, демократия и системный дизайн	65
Глава 5. Эффект второй системы	77
Глава 6. Доведение до сведения	85
Глава 7. Почему провалился вавилонский проект?	97
Глава 8. Попытки измерить	113

Глава 9. Непосильный груз	123
Глава 10. Документарная гипотеза	133
Глава 11. Планируй выбросить.....	141
Глава 12. Заточенные инструменты.....	153
Глава 13. Целое и части.....	167
Глава 14. И грянул гром	181
Глава 15. Другая сторона	193
Глава 16. Серебряной пули нет — существенные и частные признаки инженерии программного обеспечения	211
Глава 17. Повторный выстрел «Серебряной пули нет».....	245
Глава 18. Тезисы мифического человеко-месяца: false или true?	273
Глава 19. Мифический человеко-месяц двадцать лет спустя	305
Эпилог. Пятьдесят лет удивления, воодушевления и радости.....	350
Заметки и ссылки	352



Фредерик Брукс © Jerry Markatos

ОБ АВТОРЕ

Фредерик П. Брукс-младший — Кенановский профессор информатики Университета Северной Каролины в Чапел-Хилле. Наиболее известен как «отец компьютера IBM System/360», работал менеджером проекта по его разработке, а затем менеджером проекта OS/360 в IBM.

За эту работу в 1985 году он, Боб Эванс и Эрих Блох были награждены Национальной медалью США в области технологий и инноваций. Ранее был архитектором компьютеров IBM Stretch и Harvest.

В Чапел-Хилле доктор Брукс основал отдел информатики и возглавлял его с 1964 по 1984 год. Служил в Национальном научном совете и Научном совете Министерства обороны США.

ПРЕДИСЛОВИЕ НАУЧНОГО РЕДАКТОРА К РУССКОМУ ИЗДАНИЮ 2020 ГОДА

Дмитрий Мамонов,
Chief Architect, Wrike

Сколько битов в байте? Восемь.

Но так было не всегда. Ни общего термина *байт*, ни стандарта «8-бит» в 1960-х еще не было. Книга «Мифический человеко-месяц», которую вы держите в руках, была написана Фредериком Бруксом как попытка анализа и систематизации его опыта руководства в проекте IBM System/360, грандиозный коммерческий успех которого кардинально изменил всю компьютерную индустрию своего времени и, в частности, привел к формированию единого стандарта кодирования информации — байту.

Эту книгу стоит прочитать для понимания истории развития индустрии разработки программного обеспечения, но и не только поэтому. Брукс одним из первых столкнулся с трудностями разработки больших программных проектов. Да, за прошедшие с первого издания книги 45 лет в разработке произошли качественные изменения во всем, начиная от производительности компьютеров и заканчивая подходами к организации разработки, тем не менее темы, которые поднимаются в этой книге, все еще актуальны и стоят того, чтобы над ними задуматься.

Для своего времени операционная система OS/360, о которой говорит Брукс, разрабатываемая в рамках проекта System/360, была, наверное, так же широко известна среди специалистов, как сейчас Linux. Отсылки к ней не требовали отдельных пояснений. Но с тех пор сменилось уже много поколений операционных систем, и для лучшего понимания событий, описываемых в книге, стоит напомнить ключевые детали, связанные с историей этого проекта.

КРАТКАЯ ИСТОРИЯ РАЗРАБОТКИ SYSTEM/360

В начале 1960-х корпорация IBM была абсолютным лидером рынка компьютеров. Ее доля составляла 75 %. Однако перспективы становились все менее радужными. Абсолютно все системы IBM были несовместимы между собой. Серии 1401, 1620, 7070 и т. д. были полностью изолированными. Хотите перейти с 1401 на 1620 — купите не только новый процессор, но и всю периферию. Софт тоже придется переписать.

Дорогое удовольствие для клиента, не каждый на такое решится. Да и для самой IBM ситуация выглядела плохо. Приходилось поддерживать производство устаревшего оборудования, содержать штат специалистов, умеющих его настраивать, обучать инженеров на стороне клиента устаревшим технологиям. При этом переход с одной системы на другую требовал полного переобучения. Ситуацию усугубляло то, что многие из систем были узкоспециализированными, например диспетчерские системы.

И вот в январе 1961 года 29-летний Брукс представляет проект очередной серии 8000. Конечно, новая система лучше предыдущих во всем, но в одном она с ними одинакова. Это еще один полностью уникальный комплекс, переход на который обойдется клиентам в миллионы, как и его поддержка самой IBM. Понятно, что это тупик. Проект закрывают, а Бруксу предлагают возглавить груп-

пу по разработке совершенно новой системы. Но какой — никто не знал. Одно было понятно: новая система должна обеспечить в дальнейшем обратную совместимость как на аппаратном, так и на программном уровнях, а также быть системой общего назначения, подходящей и банкам, и военным, и ученым.

Была сформирована группа из 25 человек во главе с Бруксом, которая занялась разработкой плана новой системы. Процесс двигался медленно, и чтобы его ускорить, руководство решило переселить рабочую группу в отель в пригороде Нью-Йорка с ультиматумом, что команда не выйдет оттуда, пока не придет к общему решению. И они пришли. И этому решению дали зеленый свет.

Весь аппаратно-программный комплекс называли System/360, а операционную систему — OS/360. Иронично, что проблемы обратной совместимости были решены за счет отказа от совместимости с предыдущими системами.

Типичная конфигурация высокопроизводительной серии, выпущенной в 1967 году, составляла до 16,6 миллиона операций в секунду и порядка 512 килобайт оперативной памяти. Данные могли храниться на магнитных лентах, до 20 мегабайт на ленте, либо на жестких дисках объемом около 5 мегабайт. В качестве высокоуровневых языков программирования использовались Cobol, Fortran, Algol и PL/1.

Разработка системы заняла существенно больше планируемых сроков, ее стоимость составила не \$625 млн, но \$5,25 млрд — не многим меньше, чем программа Apollo с ее ракетами, астронавтами и высадкой на Луну за тот же 1965 год. Риск банкротства для IBM был вполне реален, но все обошлось. Анонс системы состоялся 7 апреля 1964 года, а первые продукты были выпущены в середине 1965-го. Коммерческий успех был грандиозный. Принцип взаимозаменяемости компонент, заложенный в рамках этой системы, соблюдается и по сей день.

Однако с организационной точки зрения проект нельзя назвать вполне удачным. Быстро, дешево, качественно — ни одно из этих свойств достигнуто не было.

Однако эта книга, как ретроспектива самого проекта, вполне успешна. Многие современные профессиональные разработчики знакомы с ней. И даже в наши дни, пусть и нечасто, отсылки на изложенные Бруксом идеи можно услышать в технической беседе. Тем не менее со времен написания книги в индустрии многое изменилось, и кажется важным подчеркнуть некоторые ключевые достижения, произошедшие в наше время.

ВЗГЛЯД ИЗ XXI ВЕКА

Вы держите в руках перевод второго издания книги 1995 года. Кроме оригинального материала первого издания, вышедшего в 1975-м, оно расширено новыми главами, в которых автор отражает свои новые и пересмотренные соображения, а также изменения в индустрии за 20 лет, прошедших с первого издания. Сейчас, спустя еще четверть века, стоит отметить ряд качественных изменений, достигнутых в наше время.

С точки зрения производительности компьютеров достигнут огромный прогресс, и не только в сравнении с 1965 годом, но и с 1995-м. При этом в наши дни динамика роста производительности процессоров существенно замедлилась. Сейчас новые системы развиваются скорее за счет расширения количества вычислительных ядер процессора, нежели за счет увеличения производительности одного ядра. Однако для разработчика современные компьютеры и ноутбуки великолепны.

Персональный компьютер, как массовое явление 90-х, существенно расширил аудиторию разработчиков программного обеспечения,

при этом подход к разработке изменился не так значительно. А вот глобальное распространение интернета привело к кардинальным изменениям. Возможность не только мгновенно получить любую доступную информацию, но и публиковать собственные материалы, общаться в реальном времени — все это принципиально изменило современный подход к организации работы, и не только в разработке. Смартфоны — это второй качественный рывок в коммуникации, произошедший за последние чуть более чем 10 лет.

Гигантские изменения произошли и в плане развития инфраструктуры разработки. Использование систем управления версиями кода, таких как `git`, `mercurial` или `svn`, стало стандартом де-факто для организации совместной работы над кодом приложения. Существует множество систем для организации непрерывной интеграции кода, сборки, регрессионного тестирования. Юнит-тестирование адаптировано индустрией, есть огромное количество зрелых фреймворков для написания автоматических тестов. Облачные среды исполнения, такие как `AWS` или `GCE`, дали возможность легко получить ресурсы, необходимые для развертывания приложения, и многое, многое другое. При этом все перечисленное доступно как сервис и до определенной степени бесплатно.

Современные платформы разработки, например `Java`, `.Net`, `Python` и прочие, и сопутствующие им библиотеки кода позволяют решать большинство прикладных задач разработки гораздо лучше, чем `C++` образца 1995 года. Конечно, и `C++` с тех пор также сильно развился. Появляются и языки, такие как `Rust`, продвигающие новые принципы программирования.

Одной из лучших сред разработки в 90-х была `Delphi`: функции автодополнения при написании кода были доступны уже тогда. Но возможность писать код чуть быстрее не так важна, как возможность легко вносить в него изменения. В 1999-м вышла книга Мартина Фаулера «Рефакторинг», и уже через несколько лет появились среды разработки, такие как `IntelliJ Idea`, поддержи-

вающие техники рефакторинга кода. Способность быстро и без ошибок вносить изменения в уже написанный код позволила адаптировать приложение к постоянным изменениям требований.

Жизненный цикл современных программных продуктов также кардинально изменился. Значительная часть современной разработки — это web-приложения, доступные как сервис. Развертывание новой версии таких приложений занимает минуты, и релиз новой версии часто происходит по несколько раз в день. Это совсем не то же самое, что релиз коробочных продуктов один раз в год. Даже в мобильной разработке новую версию приложения можно выпускать каждые несколько дней. Разработка короткими итерациями, а не проектами длительностью в год сейчас стала нормой.

Радикально новые подходы в организации разработки также стали развиваться совсем недавно. Значимым толчком в этом направлении стала публикация Манифеста Agile в 2001 году, в котором кроме четырех широко известных принципов был задан и главный посыл: *«Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим»*. И многие новые методы разработки приобрели популярность. Сейчас все больше компаний организуют разработку по Scrum, и это можно считать нормой. При этом сам подход появился сравнительно недавно, в 1995 году, а широкое применение получил уже в нашем веке.

Все перечисленное мы уже привыкли воспринимать как должное. Дети в детском саду организуют каналы на YouTube, почему нет. Перефразируя Гибсона, можно сказать, что будущее уже здесь, и оно уже почти равномерно распределено. Но в конце 80-х — начале 90-х, когда формировался материал второго издания книги Брукса, положение дел в индустрии разработки все еще было во многом сходно с историей проекта OS/360. Неудивительно, что Брукс выражал определенный скепсис в отношении того, что

в ближайшем десятилетии (от 1985 года) случится качественный прорыв, который позволит разрабатывать программное обеспечение на порядок быстрее. Но он также высказал и надежду, что на большей дистанции, в течение 40 лет, за счет множества трудоемких и локальных изменений в каждой из областей разработки прорыв все-таки случится. И оказался прав.

Тем не менее и сейчас мы продолжаем повторять многие ошибки, которые Брукс наблюдал еще полвека назад. Пусть многие темы, поднимаемые в книге, и выглядят очевидно устаревшими, но это лишь видимость. Фундаментальные проблемы, стоящие за ними, продолжают оставаться актуальными и в наше время.

АКТУАЛЬНОСТЬ ТЕМ

Наверняка многие менеджеры могут привести примеры проектов, которые были выполнены в срок, уложились в бюджет, а заказчик остался доволен. Типовые проекты существуют, и их планирование вполне реально. Но во многом разработка продолжает оставаться инновационной, а структура приложений — сложной. Такие проекты принципиально невозможно планировать: ни сроки их завершения, ни необходимые ресурсы, ни то, что будет получено в итоге. И хотя известно, что в такой ситуации лучшее, что можно сделать, это работать короткими итерациями, отслеживая прогресс и корректируя планы в конце каждого цикла, реализовать такой подход удается не всегда.

Дональд Кнут, известный математик и теоретик, продвигает идею *грамотного программирования* (literate programming). Кроме того, он автор и разработчик издательской системы TeX, являющейся стандартом подготовки научных статей. Разработку системы Кнут планировал завершить за один год, но на то, чтобы довести ее до уровня совершенства, ему потребовалось более десятилетия.

С реальными проектами так не получается. Всегда есть реальный бюджет и сроки. К сожалению, и сейчас, как и во времена Брукса, в случае отставания проекта частой реакцией руководства является добавление в проект дополнительных разработчиков. Это приводит к тому, что проект затягивается на еще больший срок, и тут дело даже не в том, что мы не понимаем последствий таких действий. Просто в безвыходной ситуации нужно сделать хоть что-то, чтобы не быть обвиненным в бездействии. А если единственный способ, с помощью которого менеджер может повлиять на горящий проект, — добавление в него больше ресурсов, именно это менеджер и сделает, можно быть уверенным.

Вопросы, связанные со сложностью планирования проектов, подробно рассматриваются в книге. Следует отметить, что рекомендации по планированию, изложенные в главах первого издания, признаны ошибочными самим автором, однако в материалах второго издания вполне четко излагаются принципы инкрементальной и итеративной разработки. Существенная часть книги посвящена вопросам концептуальной сложности приложений и их архитектуры.

Приложения, которые мы используем повседневно, такие как веб-браузер, чаты, электронные таблицы и многое-многое другое, уже достигли того уровня совершенства, что взаимодействие с ними происходит естественно и непринужденно, будто все так и должно быть. Но как разработчики программного обеспечения, мы часто сталкиваемся с не вполне восторженной реакцией пользователей на продукт, который разрабатываем. «Сложный», «запутанный», «непонятный», «бесполезный» — вот эпитеты, которые не стесняются раздавать пользователи. И хотя каждый из нас, конечно, старается сделать свою работу хорошо, подобная оценка часто бывает справедлива.

Вопрос концептуальной целостности архитектуры остается более чем актуальным. Как добиться того, чтобы разные части приложе-

ния органично стыковались между собой, а схожие задачи в различных его частях решались аналогичным образом? Как организовать интерфейс так, чтобы смысл отображаемого на экране был понятен пользователю, а реакция на выполненное действие не вызывала удивления и раздражения? В решении этих вопросов Брукс видит ключевую роль Архитектора — человека, который продумывает механику взаимодействия пользователя с системой. Сейчас такую роль часто выполняет UX-дизайнер, но суть проблем осталась та же. Как и Брукс, мы стремимся создавать целостные, удобные приложения, работать с которыми приятно и эффективно, пусть это и не всегда удастся с первого раза.

КУДА МЫ ИДЕМ

Современная индустрия переживает этап стремительного взросления. Это ярко проявляется в том, что многие компании начали вводить профессиональные уровни разработчиков по аналогии с системой разрядов на производстве. Конечно, это следствие того, что подходы ко множеству задач систематизированы, и, действуя по схеме, можно добиться успешного их решения. Но в этом же кроется и некоторая опасность. Повторяя заготовленное решение, можно упустить возможность реализовать задачу лучшим образом.

Нетрудно вообразить мрачную картину того, как все еще творческая работа разработчика превратится в рутину следования правилам и инструкциям. И в какой-то мере это правильно и неизбежно, но лишь до определенной степени. Хочется верить, что фундаментальная сложность проблем, с которыми приходится иметь дело разработчикам, всегда предоставит возможность проявить себя в поиске неординарного решения.

Какой станет разработка в ближайшие 20 лет, пока нельзя сказать с полной уверенностью. Но знать, какой она была в прошлом, опре-

деленно полезно. Фредерик Брукс дает отличное описание того, какой была индустрия 25 и 45 лет назад. Прочитать эту книгу стоит хотя бы ради того, чтобы понять, как и куда развивается индустрия разработки программного обеспечения.

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.

Посвящение из издания 1975 года

Посвящается двоим людям, особенно обогатившим мои годы в IBM:

Томасу Дж. Уотсону-младшему, чья глубокая забота о людях по-прежнему пронизывает его компанию, и Бобу О. Эвансу, чей смелый менеджмент превратил работу в приключение.

Посвящение из издания 1995 года

Посвящается Нэнси, Божьему дару для меня.

ПРЕДИСЛОВИЕ К ИЗДАНИЮ 1995 ГОДА

К моему удивлению и восторгу, книга «Мифический человеко-месяц» продолжает оставаться популярной спустя 20 лет. Тираж превысил 250 тысяч экземпляров. Меня часто спрашивают, каких мнений и рекомендаций, изложенных в 1975 году, я придерживаюсь и по сей день, а какие изменились и как. Хотя я время от времени обращался к этому вопросу в своих лекциях, давно хотелось написать эссе на эту тему.

Питер Гордон (Peter Gordon), ныне партнер издательства *Addison-Wesley*, терпеливо и услужливо работает со мной с 1980 года. Он предложил, чтобы мы подготовили юбилейное издание. Мы решили не пересматривать оригинал, а перепечатать его нетронутым (за исключением тривиальных исправлений) и дополнить более актуальными мыслями.

Глава 16 является статьей «Серебряной пули нет: существенные и частные признаки инженерии программного обеспечения» (*No Silver Bullet: Essence and Accidents of Software Engineering*), опубликованной IFIPS (Международной федерацией по обработке информации) в 1986 году, которая выросла из моего опыта председательства в Научном совете Министерства обороны США. Мои соавторы этого исследования и наш исполнительный секретарь Роберт Л. Патрик (Robert L. Patrick) оказали неоценимую помощь в моем возвращении к реальным крупным проектам по созданию программного обеспече-

ния. Статья была перепечатана в 1987 году в компьютерном журнале *IEEE Computer* и получила широкую известность.

Статья «Серебряной пули нет» оказалась провокационной. Она предсказала, что в течение десятилетия не будет создано ни одного решения, которое само по себе привело бы к повышению продуктивности разработки на порядок. До истечения десятилетия остался год; мое предсказание, похоже, сбывается. Статья «СПН» стала поводом для более оживленной дискуссии в литературе, чем было заложено в «Мифическом человеко-месяце». В связи с этим в главе 17 даются комментарии по некоторым опубликованным критическим замечаниям и обновляются мнения, изложенные в 1986 году.

Работая над обзором и обновлением книги «Мифический человеко-месяц», я поразился, как мало тезисы, заявленные в ней, были подвергнуты критике, доказаны или опровергнуты текущими исследованиями и опытом в инженерии ПО. Теперь для меня оказалось полезным каталогизировать эти тезисы в сырой форме, лишенной подтверждающих аргументов и данных. В надежде, что эти голые утверждения привлекут аргументы и факты для доказательства, опровержения, обновления или уточнения, я включил этот план в главу 18.

Глава 19 как таковая является новым эссе. Читателя следует предупредить о том, что новые мнения далеко не так прочно базируются на опыте работы «в полях», как это было в оригинальной книге. Дело в том, что в последнее время я работал в университетской среде, а не в промышленности, и над небольшими, а не крупномасштабными проектами. С 1986 года я занимался инженерией ПО только как преподаватель, а не как исследователь. Мои исследования, скорее, были посвящены виртуальным средам и их приложениям.

При подготовке этой ретроспективы я стремился изложить текущие взгляды моих друзей, которые фактически работают в инженерии программного обеспечения. Я в долгу перед Барри Бемом,

Кеном Бруксом, Диком Кейсом, Джеймсом Коггинсом, Томом Демарко, Джимом Маккарти, Дэвидом Парнасом, Эрлом Уилером и Эдвардом Йордоном за их замечательную готовность делиться мнениями, вдумчиво комментировать наброски и перевоспитывать меня. Фэй Уорд великолепно справилась с изданием новых глав.

Я благодарю Гордона Белла, Брюса Бьюкенена, Рика Хейса-Рота, моих коллег из целевой группы Научного совета Министерства обороны США по военному программному обеспечению, и в особенности Дэвида Парнаса, за их существенные и стимулирующие идеи, а также Ребекку Бирли за подготовку статьи, напечатанной здесь в качестве главы 16. Анализ задачи разработки программного обеспечения в категориях *существенных* и *частных* признаков возник благодаря Нэнси Гринвуд Брукс, которая использовала такой анализ в статье о методе Судзуки преподавания игры на скрипке.

Традиции издательства Addison-Wesley не позволили мне в предисловии к изданию 1975 года выразить свою признательность сотрудникам издательства, которые сыграли ключевую роль. Следует особо отметить вклад двух человек: Нормана Стэнтона, который в то время был главным редактором, и Герберта Боэса, который в то время был художественным редактором. Боэс разработал элегантный стиль, который один рецензент, в частности, охарактеризовал как «широкие поля (и) образное использование шрифта и макета». Что еще важнее, так это его рекомендации по оформлению книги: именно он рекомендовал снабдить рисунком каждую главу. (В то время у меня было только два рисунка: Смоляные ямы и Реймский собор.) На то, чтобы найти все картинки, мне потребовался целый год, но я бесконечно благодарен за совет.

Soli Deo Gloria — одному Господу слава.

Ф. Б.-мл.

Чапел-Хилл, Северная Каролина

Март 1995

ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ

Во многих отношениях управление крупным проектом по разработке программного обеспечения похоже на управление любым другим крупным предприятием — в большей степени, чем считают большинство программистов. Но во многих других отношениях оно отличается — в большей степени, чем ожидают большинство профессиональных менеджеров.

Профессиональные знания в этой области накапливаются. Было проведено несколько конференций, сессий на конференциях AFIPS, вышло несколько книг и статей. Но они ни в коем случае еще не готовы к какой-либо систематической хрестоматийной обработке. Однако представляется уместным предложить эту небольшую книгу, отражающую, по существу, личный взгляд.

Хотя я изначально вырос на «программной» стороне компьютерных наук, в течение тех лет (1956–1963), когда была разработана автономная управляющая программа и компилятор высокоуровневого языка, я главным образом был вовлечен в аппаратную архитектуру. Когда в 1964 году я стал менеджером проекта Operating System/360, я обнаружил, что за последние несколько лет мир программирования сильно изменился благодаря прогрессу.

Опыт управления проектом OS/360 стал очень познавательным, хотя и очень разочаровывающим. Команде, включая Ф. М. Трапнелла (F. M. Trapnell), сменившего меня на посту менеджера проекта, есть чем гордиться. Указанная система обладает большими преимуществами дизайна и исполнения, и она была успешной в достижении ее широкого использования. Некоторые идеи, наиболее заметно зависящие от устройств ввода-вывода и управления внешними библиотеками, были технически инновационными и сейчас широко копируются. Теперь система является достаточно надежной, достаточно эффективной и очень универсальной.

Однако эти усилия нельзя назвать полностью успешными. Любой пользователь OS/360 быстро осознает, что до идеала еще далеко. Недостатки в дизайне и исполнении в особенности свойственны управляющей программе, в отличие от компиляторов языков. Большинство этих недостатков относятся к периоду дизайна 1964–1965 годов и, следовательно, должны быть возложены на меня. Кроме того, продукт опоздал, он занял больше памяти, чем планировалось, затраты оказались в несколько раз выше оценочных, и он не очень хорошо работал как в первом релизе, так и в нескольких последующих.

После ухода из IBM в 1965 году и приезда в Чапел-Хилл, как было первоначально согласовано, я возглавил работу над проектом OS/360 и начал анализировать опыт работы с OS/360 для того, чтобы увидеть, какие могут быть извлечены управленческие и технические уроки. В частности, я хотел бы объяснить совершенно разный опыт руководства в разработке аппаратного обеспечения для System/360 и программного обеспечения для OS/360. Эта книга — запоздалый ответ на мучительные вопросы Тома Уотсона, почему разработкой ПО трудно управлять.

В этом стремлении я извлек пользу из долгих бесед с Р. П. Кейсом (R. P. Case), помощником менеджера в 1964–1965 годах, и Трапнеллом, менеджером в 1965–1968 годах. Я сравнил свои выво-

ды с выводами других менеджеров крупных проектов, включая Ф. Дж. Корбатто (F. J. Corbato) из Массачусетского технологического университета, Джона Харра (John Harr) и В. Высоцкого (V. Vyssotsky) из *Bell Telephone Laboratories*, Чарльза Портмана (Charles Portman) из *International Computers Limited*, А. П. Ершова из вычислительной лаборатории Сибирского отделения Академии наук СССР и А. М. Пьетрасанту (A. M. Pietrasanta) из *IBM*.

Мои собственные выводы воплощены в следующих далее эссе. Они предназначены для профессиональных программистов, профессиональных менеджеров, и в первую очередь для профессиональных менеджеров-программистов.

Несмотря на то что главы написаны как отдельные эссе, в них, в особенности в главах 2–7, содержится центральный аргумент. Если кратко, то я считаю, что крупные программные проекты страдают от проблем менеджмента, отличных от проблем менеджмента мелких проектов, что обусловливается разделением труда. Я считаю, что критически необходимо сохранять концептуальную целостность самого продукта. В этих главах рассматриваются трудности достижения целостности и методы достижения этой целостности. В последующих главах рассматриваются другие аспекты менеджмента в инженерии программного обеспечения.

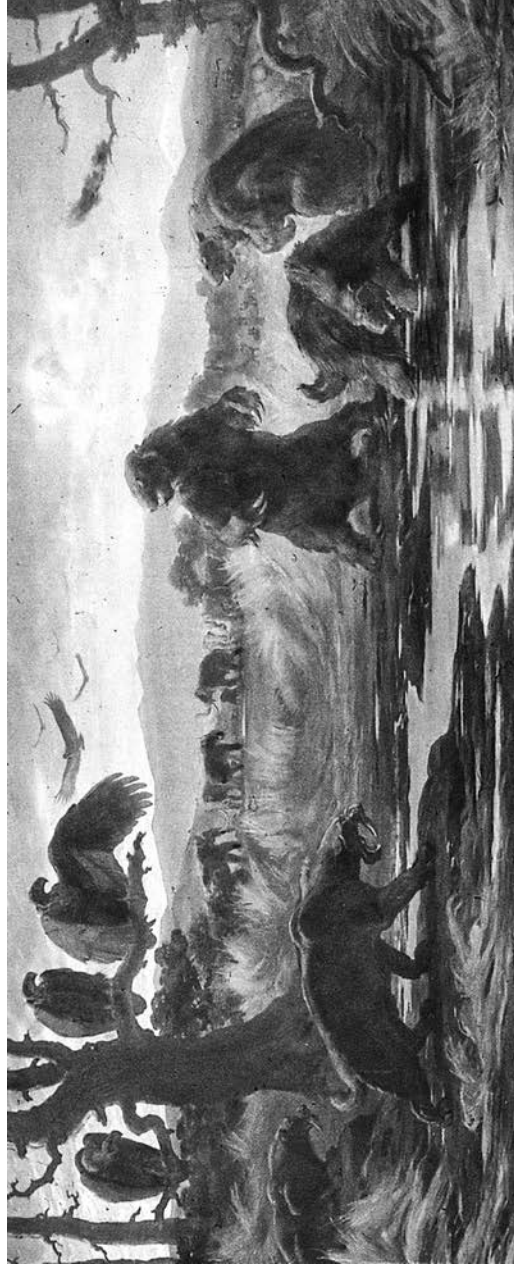
Литература в указанной области немногочисленна, но при этом весьма распылена. Поэтому я попытался дать ссылки, которые будут освещать конкретные места и направлять заинтересованного читателя к другим полезным работам. Многие мои друзья читали рукопись, и некоторые подготовили обширные полезные комментарии; там, где они казались ценными, но не вписывались в основной текст, я включил их в заметки.

Поскольку эта книга состоит из эссе и не является сплошным повествованием, все ссылки и заметки были вынесены в конец, и читателю можно пропустить их при первом чтении.

Я глубоко признателен мисс Саре Элизабет Мур, мистеру Дэвиду Вагнеру и миссис Ребекке Бернс за помощь в подготовке рукописи, а также профессору Джозефу Слоуну за советы по подбору иллюстраций.

Чапел-Хилл, Северная Каролина

Октябрь 1974 года



Ч. Р. Найт (C. R. Knight). Панно «Смоляные ямы в Ла-Брея», Палеонтологический музей
находок Джорджа К. Пейджа на ранчо Ла-Брея

1

СМОЛЯНЫЕ ЯМЫ

Корабли на мели — моряку маяк.

Голландская пословица

Ни одна из доисторических сцен не была столь яркой, как сцена смертельной схватки огромных зверей в смоляных ямах*. Мы видим, как динозавры, мамонты и саблезубые тигры борются с тисками смолы. Чем яростнее борьба, тем глубже смола затягивает, и каждому зверю, неважно, как силен он и ловок, в конце концов уготована гибель.

Разработка больших программных систем в течение последнего десятилетия была такой смоляной ямой, огромные и могущественные звери ожесточенно пытались в ней выжить. Большинство из них вышли с работающими системами, не многие достигли целей, придерживаясь графиков и бюджетов. Крупные и малые, массивные или жилистые, команда за командой вязли в смоле. Отдельно взятая проблема, по-видимому, не вызывает никаких трудностей — любую конкретную лапу можно вырвать из тисков. Но накопление одновременных и взаимодействующих факторов постепенно замедляет движение. Все, похоже, были удивлены вязкостью проблемы, и ее природу трудно было понять. Но мы должны попытаться понять, если хотим ее решить.

Поэтому давайте начнем с определения ремесла системного программирования и присущих ему радостей и горестей.

СИСТЕМА ПРОГРАММИРОВАНИЯ КАК ПРОДУКТ

Иногда в новостях можно прочесть о том, как два программиста в перестроенном гараже создали важную программу, которая пре-

* Смоляные ямы Ла-Брея — месторождение окаменелостей верхнего плейстоцена, расположенное в парке Хэнкок, Лос-Анджелес. Наличие смоляных ям позволило обнаружить животных, сохранившихся в этом материале. Смола медленно поднималась на поверхность земли в этом регионе в течение нескольких десятков тысяч лет, образуя сотни вязких углублений, которые заключали в ловушку животных, имевших неосторожность туда подойти. — *Примеч. пер.*

восходит все усилия крупных команд. И каждый программист готов поверить в такие истории, ибо знает, что может собрать любую программу гораздо быстрее, чем та тысяча инструкций языка в год, о которых сообщается, будто они пишутся в производственных командах.

Почему же тогда не все команды по программированию были заменены специализированными гаражными дуэтами? Нужно смотреть на то, что производится.

В левом верхнем углу рис. 1.1 находится *программа*. Как таковая она закончена, готова к выполнению автором в системе, в которой



Рис. 1.1. Эволюция продукта системы программирования

была разработана. Это *то*, что обычно производится в гаражах, причем является объектом, который отдельный программист использует для оценивания продуктивности.

Существует два способа конвертирования программы в более полезный, но более дорогостоящий объект. Эти два способа представлены по краям рисунка.

Двигаясь вниз через горизонтальную границу, программа становится *программным продуктом*. Это программа, которая может выполняться, тестироваться, исправляться и расширяться кем угодно. Она может использоваться во многих операционных средах, для многих наборов данных. Для того чтобы стать публичным программным продуктом, программа должна быть написана в обобщенном виде. В частности, диапазон и форма входных данных должны быть обобщены настолько, насколько это позволяет базовый алгоритм. Затем программа должна быть тщательно протестирована, чтобы сделать ее надежной. Для этого нужно подготовить достаточное количество контрольных примеров для проверки диапазона допустимых значений входных данных и определения его границ, обработать эти примеры и зафиксировать результаты. Наконец, продвижение программы до стадии программного продукта требует ее тщательного документирования, с тем чтобы любой мог ею пользоваться, исправлять и расширять. В качестве эмпирического правила я оцениваю, что программный продукт стоит по крайней мере в три раза больше, чем отлаженная программа с той же функциональностью.

Двигаясь через вертикальную границу, программа становится компонентом *системы программирования*. Он представляет собой коллекцию взаимодействующих программ, скоординированных по функциональности и упорядоченных по формату таким образом, что их совокупность представляет собой целостное обеспечение для выполнения крупных задач. Для того чтобы стать компонентом системы программирования, программа должна

быть написана так, чтобы каждый вход и выход подчинялся синтаксису и семантике с точно определенными интерфейсами. Программа также должна быть составлена так, чтобы использовать только прописанный бюджет ресурсов, включая пространство памяти, устройства ввода-вывода, компьютерное время. Наконец, программа должна быть протестирована вместе с другими компонентами системы во всех ожидаемых комбинациях. Это тестирование получится обширным, так как число случаев растет комбинаторно. Указанное тестирование является времязатратным, поскольку трудноуловимые ошибки возникают из-за неожиданных взаимодействий отлаживаемых компонентов. Компонент системы программирования стóит по меньшей мере в три раза дороже, чем автономная программа с той же функциональностью. Стоимость может быть больше, если система имеет много компонентов.

В правом нижнем углу рис. 1.1 находится *продукт систем программирования*. Он отличается от простой программы во всех вышеперечисленных отношениях. Он стóит в девять раз дороже. Но он является поистине полезным объектом, конечным результатом наибольших усилий в разработке системных программных продуктов.

РАДОСТИ РЕМЕСЛА

Почему программирование приносит радость? Какие удовольствия в качестве награды ждут его практика?

Во-первых, это чистая радость от изготовления вещей. Как ребенок получает удовольствие от лепки куличика из песка, так и взрослый наслаждается созданием вещей, в особенности вещей по своему собственному дизайну. Думаю, что этот восторг должен быть образом восторга Господа от созидания, восторга, проявля-

ющегося в отчетливости и новизне каждого листика и каждой снежинки.

Во-вторых, это удовольствие делать вещи, которые полезны другим людям. Глубоко внутри мы хотим, чтобы результат нашего труда был полезен и востребован. В этом отношении система программирования практически не отличается от первой детской подставки для ручек из глины «для папиного кабинета».

В-третьих, это увлекательная возможность придавать форму сложным головоломным объектам, состоящим из взаимосвязанных движущихся частей, и наблюдать за тем, как они работают в тонких циклах, воспроизводят последствия принципов, заложенных с самого начала. Запрограммированный компьютер обладает всей прелестью пинбольной машины или музыкального автомата, доведенных почти до предела совершенства.

В-четвертых, это радость от постоянного усвоения нового — этого предостаточно в неповторяющейся работе. Так или иначе, проблема всегда нова, и решающий ее чему-то учится: иногда практическому, иногда теоретическому, а иногда и тому и другому.

Наконец, это удовольствие работать в такой послушной среде. Программист, как и поэт, работает лишь в небольшом отрыве от чистого мыслительного материала. Он строит свои замки из воздуха, творя усилием воображения. Немногие среды творения настолько гибки, так легко поддаются полировке и переработке, так легко способны реализовывать грандиозные концептуальные структуры. (Как мы увидим позже, у этой самой сговорчивости есть свои проблемы.)

И все же программный конструкт, в отличие от слов поэта, реален в том смысле, что он движется и работает, производя видимые результаты, отличные от него самого. Он печатает результаты, рисует картинки, издает звуки, двигает манипуляторами. В наше время

волшебство мифов и легенд стало реальностью. Стоит набрать правильное заклинание на клавиатуре, и экран дисплея оживает, показывая вещи, которых никогда не было и не могло быть.

Таким образом, программирование доставляет удовольствие, поскольку отвечает глубокой внутренней потребности в творчестве и удовлетворяет чувственные потребности, которые есть у всех нас.

ГОРЕСТИ РЕМЕСЛА

Однако не все является наслаждением, и знание присущих ремеслу горестей облегчает их ношу.

Во-первых, человек должен действовать безупречно. Компьютер похож на магию и в этом отношении тоже. Если один символ, одна пауза заклинания не находится строго в правильной форме, то волшебство не работает. Люди не совершенны, и немногие сферы человеческой деятельности этого требуют. Необходимость приспособливаться к требованиям совершенства — это, я думаю, самая трудная часть овладения программированием.¹

Конечные целевые критерии устанавливают другие люди, они предоставляют ресурсы и обеспечивают информацией. Редко кто-то контролирует условия своей работы или даже ее цель. С точки зрения менеджмента, мы зависим от многих. Однако, похоже, в любой работе формальная власть несоизмерима с ответственностью. На практике фактические (в отличие от формальных) полномочия приобретаются с самого момента достижения конечной цели.

Зависимость от других имеет особенно неприятную системную сторону. Он зависит от чужих программ, которые ча-

сто плохо спроектированы, плохо имплементированы, поставлены не полностью (нет исходного кода или тестовых случаев) и плохо документированы. Программист должен тратить часы на изучение и исправление того, что в идеале должно быть завершенным, доступным и пригодным для использования.

Следующая горесть состоит в том, что заниматься дизайном великих концепций — это удовольствие; отыскивать мелкие ошибки — это просто работа. Вместе с любой творческой деятельностью приходят тоскливые часы монотонного, кропотливого труда, и программирование не является исключением.

Далее обнаруживается, что отладка имеет линейную сходимость или даже хуже, где ожидалось что-то вроде квадратичного подхода к концу. И поэтому тестирование затягивается, при этом на поиск последних ошибок приходится тратить больше времени, чем на поиск первых.

И последней каплей, которая может переполнить чашу горестей, может стать опасность того, что продукт, над которым так долго трудились, устареет еще до его завершения. Коллеги и конкуренты уже вовсю гоняются за новыми и более совершенными идеями. И уничтожение плода вашей мысли уже не только задумано, но и запланировано.

Это звучит хуже, чем есть на самом деле. Новый и более совершенный продукт, в общем-то, *недоступен*, когда кто-то завершает свой собственный; о нем только говорят. И к тому же на его разработку уйдут месяцы. Настоящий тигр никогда не сравнится с бумажным, только если не потребуются его реальное использование. И тогда добродетели реальности получают удовлетворение сами по себе.

Разумеется, технологии, которые мы используем, *постоянно* развиваются. Как только дизайн замораживается, он становится устаревшим с точки зрения его концепций. Но имплементация

реальных продуктов требует разделения на фазы и подгруппы. Устаревание имплементации должно измеряться по отношению к другим существующим имплементациям, а не по отношению к нереализованным концепциям. Задача и миссия состоят в том, чтобы найти реальные решения реальных задач в реальные сроки с имеющимися ресурсами.

Это и есть программирование и смоляная яма, в которой вязнут многие усилия и творческая деятельность со своими собственными радостями и горестями. Для многих радости намного перевешивают горести, и для них остальная часть этой книги попытается проложить дощатый мостик через смоляную яму.

Restaurant Antoine



AVIS au Public

La bonne Cuisine exige le temps necessaire a sa propre preparation. Si on vous fait attendre, c'est pour mieux vous servir, et vous plaire.

ENTREES (Continuation)

Côtelettes d'Agneau Grillées .90	Côtelettes d'Agneau Maison d'Or 1.25
Côtelettes d'Agneau Champignons Frais 1.50	Côtelettes d'Agneau Parisienne 1.25
Filet de Boeuf Nature 1.75	Tournedos Nature 1.25
Entrecote de Boeuf Minute 1.25	Entrecote de Boeuf Nature 1.75
Filet de Boeuf Béarnaise 2.00	Entrecote de Boeuf Marchand de Vin 2.00
Tournedos Medici 1.50	Filet de Boeuf à La Hawaïan 2.00
Tournedos Béarnaise 1.50	Tournedos Marchand de Vin 1.50
Filet de Boeuf Champignons Frais 2.25	Tournedos Hawaïan 1.50
Rix de Veau à La Financière .90	Brochette de Foie de Volaille .90
Pigeonneaux Sauce Paradis 1.50	Pigeonneaux Grille 1.25
Tripe à La Mode de Caen (Order in advance) .90	Chateaubriand (30 Minutes) for two 3.75

LEGUMES

Epinauds à La Crème .40	Petit Pois à La Française .40
Broccoli Hollandaise .60	Chouxfleur au Gratin .40
Haricots au Beurre .40	Asperges Fraiches au Beurre .40
Pomme au Gratin .40	Carottes à La Crème .40
Pommes de Terre Soufflées .40	

SALADES

Salade à La Antoine .30	Fond d'Artichaut Bayard .60
Salade Mirabeau .40	Salade Laitue et Oeufs .40
Salad Laitue et Roquefort .50	Tomate Frappée à La Jules Caesar .40
Salade de Legumes .40	Salade Coeur de Palmier .75
Avocat Antoinette .50	Points d'Asperges .40
Salade d'Anchois .75	Avocat Vinaigrette .35
Laitue and Tomates .35	

DESSERTS

Gateau Moka .25	Glace à La Vanille .20
Meringue Glacée .25	Crêpes à La Gelée .50
Les Crêpes Suzette .75	Omelette au Rhum .75
Fruits de Saison au Cognac .35	Cerises Jubilee .75
Soufflee au Chocolat (For Two) 1.50	Glace Sauce Chocolat .35
Omelette Alaska à La Antoine (For Two) 1.50	Fraises au Kirsch .50
Omelette Soufflée à La Jules Caesar (For Two) 1.00	Soufflée à La Vanille (For Two) 1.50

FROMAGES

Roquefort .30	Liederkranz .30	Gruyere .30
Camembert .40		Crème de Philadelphie .30

CAFE

Grand Café .15	Café au Lait .15	Thé .15
Café Brûlot Diabolique .60	The Glacée .15	Demi-Tasse .10

EAUX MINERALES—BIERE—CIGARS—CIGARETTES

White Rock .30	Biere Local .20	Cigars
Perrier .50	Biere du Pays .25	Cigarettes
Vichy .50		



Roy L. Hicatore, Proprietor

713-717 St. Louis Street

New Orleans, La.

Фото меню ресторана «Antoine», Новый Орлеан.
UPI Photo/Архив Беттмана

2

МИФИЧЕСКИЙ ЧЕЛОВЕКО-МЕСЯЦ

Высокая кухня требует времени. Если вас заставляют ждать, то только для того, чтобы лучше вас обслужить и доставить вам удовольствие.

*Меню ресторана «Antoine»,
Новый Орлеан*

Программные проекты чаще проваливаются из-за нехватки календарного времени, чем по всем остальным причинам вместе взятым. Почему такая причина катастрофы наиболее распространена?

Во-первых, наши методики оценивания развиты слабо. По сути, они отражают молчаливое допущение, что все пойдет как надо.

Во-вторых, наши методики оценивания ошибочно путают усилия с прогрессом, допуская, что люди и месяцы взаимозаменяемы.

В-третьих, поскольку мы не уверены в наших оценках, менеджерам часто не хватает вежливого упрямства шеф-повара ресторана *Antoine*.

В-четвертых, ход выполнения графика плохо контролируется. Методы, проверенные и рутинные в других инженерных дисциплинах, считаются радикальными инновациями в инженерии программного обеспечения.

В-пятых, когда выявляется отставание от графика, естественным (и традиционным) ответом становится добавление рабочей силы. Это все равно что подлить масла в огонь: будет только хуже, гораздо хуже. Больше огня требует больше горючего, и в результате начинается регенеративный цикл, который заканчивается катастрофой.

Контроль графика работ станет темой отдельного эссе. Рассмотрим другие аспекты проблемы подробнее.

ОПТИМИЗМ

Все программисты — оптимисты. Возможно, это современное колдовство в особенности привлекает тех, кто верит в счастливый конец и сказочных волшебниц. Возможно, сотни мелких разочарований отпугивают всех, кроме зацикленных по привычке

на конечной цели. Возможно, дело просто в том, что компьютеры существуют не так много лет, программисты — еще моложе компьютеров, а молодые всегда оптимистичны. Но как бы ни работал процесс отбора, результат будет бесспорным: «на этот раз она обязательно запустится» или «я только что отыскал последний баг».

Таким образом, первое ложное допущение, лежащее в основе определения графика разработки, состоит в том, что *все пойдет хорошо, то есть каждая часть работы будет продолжаться столько, сколько она «должна» длиться*.

Повсеместное присутствие оптимизма среди программистов заслуживает пристального изучения. Дороти Сэйерс (Dorothy Sayers) в своей замечательной книге «Ум творца» (*The Mind of the Maker*) делит творческую деятельность на три стадии: идея, имплементация и взаимодействие. Таким образом, книга, компьютер или программа сначала возникают как идеальный конструкт, построенный вне времени и пространства, но завершенный в сознании автора. Он реализуется во времени и пространстве с помощью пера, чернил и бумаги или с помощью проводов, кремния и феррита. Творение завершается, когда кто-то читает книгу, пользуется компьютером или выполняет программу, тем самым взаимодействуя с умом творца.

Описание, которое использует мисс Сэйерс, может пролить свет не только на человеческую творческую деятельность, но и на христианское учение о Троице, и будет полезно в нашей нынешней работе. Для людей, создающих что-то, незавершенность и противоречивость идей становятся ясными только в процессе имплементации. Таким образом, написание, экспериментирование, «отработка» являются существенными дисциплинами для теоретика.

Во многих творческих видах деятельности среда исполнения является неподатливой. Древесина раскалывается, краски размазывают-

ся, электрические цепи закольцовываются. Эти физические лимиты среды ограничивают круг идей, которые могут быть воплощены, и они также создают неожиданные трудности в имплементации.

Имплементация, таким образом, требует времени и пота как по причине физических сред, так и по причине неадекватности лежащих в основе идей. Мы склонны обвинять физические среды в большинстве наших трудностей с имплементацией, поскольку они чужды нам — в отличие от идей, которыми мы гордимся.

Однако компьютерное программирование создает продукт с помощью чрезвычайно податливой среды. Программист строит из чистого мысленного материала — концепций и их очень гибких представлений. Поскольку среда является податливой, мы не ожидаем больших трудностей в имплементации; отсюда наш всепроникающий оптимизм. Поскольку наши идеи могут исходить из ложных умозаключений, в нас самих есть баги; так что наш оптимизм не является оправданным.

В отдельно взятой работе допущение о том, что все пойдет хорошо, влечет за собой влияние вероятностей на график работ. Возможно, все действительно пойдет **по плану**, так как существует вероятностное распределение для задержки, которая будет неожиданно встречена по ходу, а «отсутствие задержки» имеет конечную вероятность. Однако большие проекты в разработке состоят из многих задач, и некоторые из них выстроены впритык. Вероятность того, что все пойдет хорошо, становится ничтожно малой.

ЧЕЛОВЕКО-МЕСЯЦ

Второй ложный образ мышления выражается самой единицей измерения, используемой при оценивании и определении графика работ: человеко-месяц. Стоимость действительно варьируется

в зависимости от числа людей и числа месяцев. Ход выполнения — нет. *Следовательно, человеко-месяц как единица измерения объема работы является опасным и обманчивым мифом.* Из него вытекает, что люди и месяцы взаимозаменяемы.

Люди и месяцы являются взаимозаменяемыми только тогда, когда работа может быть распределена между многими работниками без коммуникаций между ними (рис. 2.1). Это верно для жатвы пшеницы или сбора хлопка, но даже приблизительно неверно для программирования систем.

Когда разделение работы невозможно из-за ограничений в последовательности, ее выполнения не влияет на график (рис. 2.2). Вынашивание ребенка занимает девять месяцев, независимо от того, сколько женщин поставлено выполнять задачу. Многие работы по разработке ПО имеют эту характеристику из-за последовательной природы отладки.

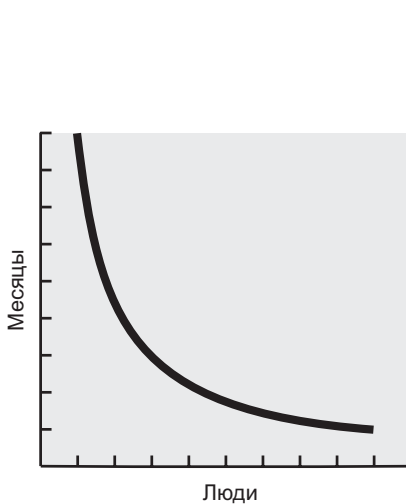


Рис. 2.1. Время относительно числа работников — идеально разделяемая работа

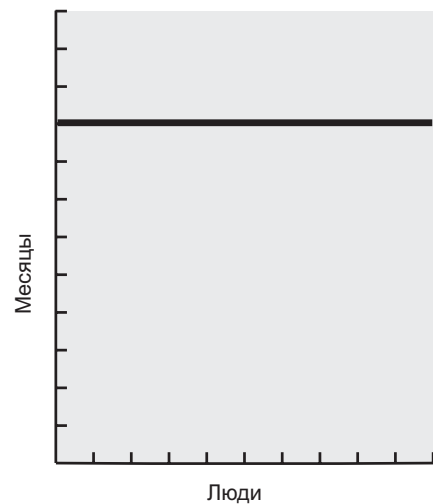


Рис. 2.2. Время относительно числа работников — неразделимая работа

В работах, которые могут быть разделены на части, но которые требуют коммуникаций между частями работы, усилие по коммуникации должно быть добавлено в объем выполняемой работы. Следовательно, даже в лучшем случае замена людей на месяцы не будет пропорциональной (рис. 2.3).

Добавленное время коммуникации состоит из двух частей: обучения и обмена данными. Каждый работник должен быть обучен технологии, целям работы, совокупной стратегии и плану работы. Это обучение нельзя разбить на части, поэтому данная часть затрат изменяется линейно в зависимости от числа занятых.¹

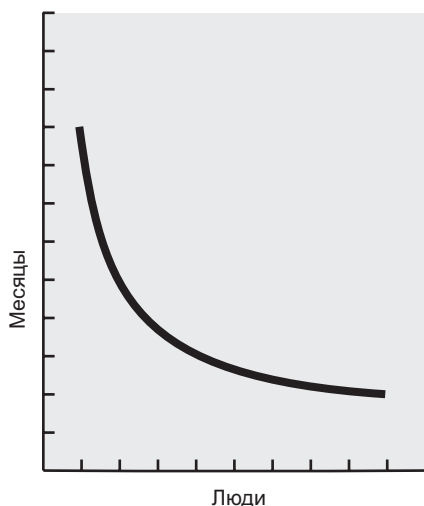


Рис. 2.3. Время относительно числа работников — делимая работа, требующая коммуникации

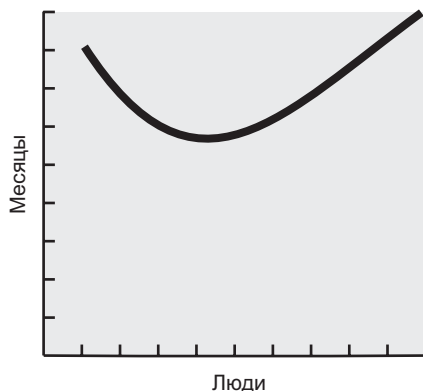


Рис. 2.4. Время относительно числа работников — работа со сложными взаимосвязями

С обменом данными дело сложнее. Если все части задания должны быть отдельно скоординированы между собой, то усилие увеличивается как $n(n-1)/2$. Три работника требуют в три раза больше попарного обмена данными, чем два; четыре требуют в шесть раз

больше, чем два. Если, кроме того, необходимо проводить конференции между тремя, четырьмя работниками и т. д. в целях совместного решения задачи, то дела становятся еще хуже. Добавленное усилие по коммуникации может полностью нивелировать разделение исходной работы и привести нас к ситуации, изображенной на рис. 2.4.

Поскольку разработка ПО, в сущности, является системным усилием — упражнением в сложных взаимосвязях: коммуникационное усилие становится огромным и быстро начинает преобладать над усилиями по уменьшению времени отдельных частей работы. И тогда добавление большего числа людей удлинняет, а не сокращает график.

ТЕСТИРОВАНИЕ СИСТЕМЫ

Из всех элементов графика работ тщательному воздействию последовательных ограничений подвергаются отладка компонентов и тестирование системы. Более того, требуемое время зависит от числа и тонкости обнаруживаемых ошибок. Теоретически это число должно быть равно нулю. Мы оптимистично ожидаем, что число ошибок будет меньше, чем оказывается на самом деле. Следовательно, тестирование обычно является самой неправильно спланированной частью программирования.

В течение нескольких лет я успешно использовал следующее эмпирическое правило для определения графика работ по сборке ПО:

- $\frac{1}{3}$ планирование;
- $\frac{1}{6}$ написание кода;
- $\frac{1}{4}$ тестирование компонентов и раннее тестирование системы;
- $\frac{1}{4}$ тестирование системы, все компоненты вместе.

Оно отличается от обычного определения графика в нескольких важных отношениях:

1. Доля, посвященная планированию, имеет бóльший размер, чем обычно. Тем не менее этого едва достаточно для того, чтобы создать подробную и основательную спецификацию, и недостаточно для того, чтобы провести исследование или изучение совершенно новых технических решений.
2. *Часть* графика, посвященная отладке завершенного кода, намного больше, чем обычно.
3. Той части, которую легко оценить, то есть кодированию, выделяется только одна шестая часть графика.

Анализируя проекты, график которых был составлен привычным образом, я обнаружил, что немногие из них отводили по графику половину времени на отладку, но на практике в большинстве случаев тратили на нее половину фактического времени. Многие проекты укладывались в график на всех этапах, исключая тестирование.²

Неспособность выделять достаточный объем времени, в частности на тестирование системы, особенно катастрофична. Поскольку сбой происходит в конце графика, никто не подозревает о проблемах с графиком почти вплоть до даты поставки. Плохие новости, запоздалые и неожиданные, тревожат клиентов и менеджеров.

Более того, задержка в этом месте имеет необычайно серьезные финансовые и психологические последствия. Проект укомплектован персоналом полностью, и затраты в расчете на день являются максимальными. Что более серьезно, программное обеспечение необходимо для реализации других бизнес-задач (доставки компьютеров, эксплуатации нового аппаратного обеспечения и т. д.), и откладывание их при наступающих сроках поставки увеличивает вторичные затраты. Действительно, эти вторичные затраты могут

значительно перевешивать все остальные. Поэтому в исходном графике работ очень важно предусмотреть достаточное время на тестирование системы.

БЕЗВОЛЬНОЕ ОЦЕНИВАНИЕ

Заметьте, что для программиста, как и для шеф-повара, нетерпеливость со стороны заказчика может привести к изменению запланированного срока завершения задачи, но его недостаточно для фактического завершения. Мило, когда вам обещают приготовить омлет за две минуты. Но когда две минуты прошли, а омлет еще не готов, у клиента остается два варианта — продолжать ждать либо съесть его сырым. У заказчиков программного обеспечения всегда есть такой же выбор.

Впрочем, повар может предложить еще вариант — добавить огня. В результате часто омлет уже ничем не спасешь: подгорит с одной стороны и будет сырым с другой.

Сейчас я не думаю, что менеджерам по программному обеспечению присущи меньшие мужество и твердость, чем шеф-поварам, равно как и другим инженерам. Но составление ложного графика в угоду желаемой заказчику дате в нашей дисциплине случается гораздо чаще, чем в других областях инженерного дела. Очень трудно дать энергичную, правдоподобную и рискованную для своей работы оценку, которая не является следствием ни одного количественного метода, подкреплена малым количеством данных и основана главным образом на интуиции менеджеров.

Очевидно, что необходимы два решения. Нам нужно разработать и опубликовать показатели продуктивности, показатели встречаемости ошибок, правила оценивания и т. д. Индустрия только извлечет выгоду из обмена такими данными.

Пока методы оценивания не получают более прочной основы, менеджерам остается только мужаться и защищать свои прогнозы, настаивая, что полагаться на их слабую интуицию все же лучше, чем основываться на одних желаниях.

КАТАСТРОФА СРЫВА ГРАФИКА РАБОТ

Что делать, когда важный проект по сборке ПО отстает от графика? Добавить рабочую силу, конечно же. Как показывают рис. 2.1–2.4, это может помочь, а может и нет.

Давайте рассмотрим пример.³ Предположим, что работа оценивается в 12 человеко-месяцев и поручается трем исполнителям в течение 4 месяцев и что существуют измеримые контрольные точки A, B, C, D, которые, согласно графику, должны выпадать на конец каждого месяца (рис. 2.5).

Теперь предположим, что первая контрольная точка достигнута только по прошествии 2 месяцев (рис. 2.6). Какие варианты есть у менеджера?

1. Допустим, что работа должна быть выполнена вовремя. Будем считать, что только первая часть работы была оценена неправильно, поэтому рис. 2.6 рассказывает историю точно. Тогда остается 9 человеко-месяцев усилий и 2 месяца, так что потребуется $4\frac{1}{2}$ человека. Добавить двух человек к уже назначенным трем.
2. Допустим, что работа должна быть выполнена вовремя. Будем считать, что вся оценка была равномерно заниженной, в результате чего рис. 2.7 описывает реальную ситуацию. Тогда остается 18 человеко-месяцев усилий и два месяца, и поэтому потребуется 9 человек. Добавить 6 человек к уже назначенным трем.

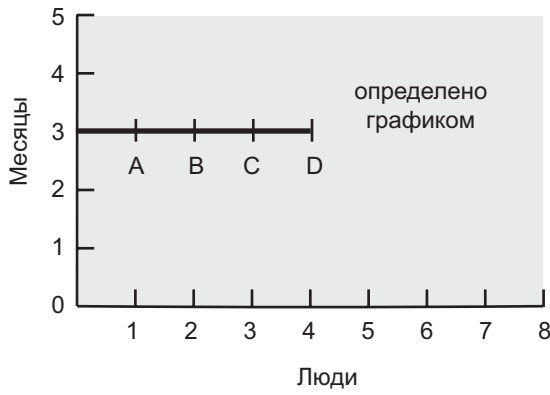


Рис. 2.5

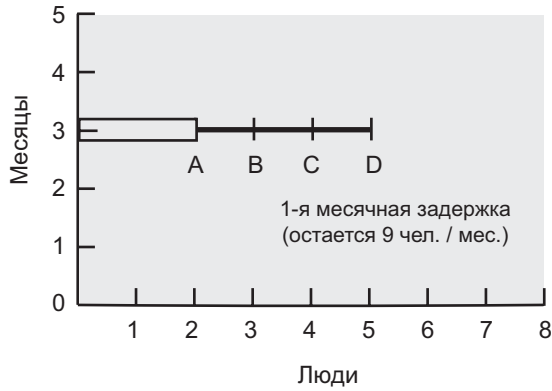


Рис. 2.6

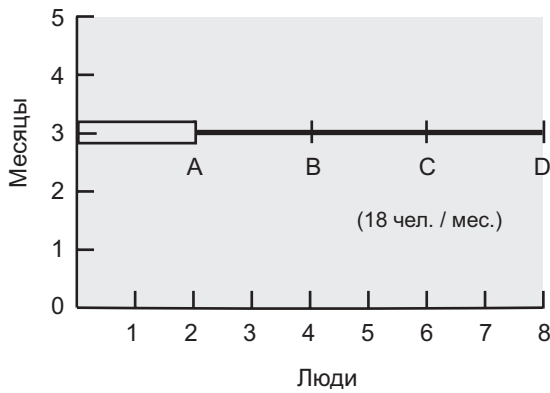


Рис. 2.7

3. Изменить график. Мне нравится совет, данный П. Фэггом (P. Fagg), опытным инженером аппаратного обеспечения: «Маленьких задержек не бывает». Он предлагает выделить в новом графике достаточно времени на тщательное и внимательное выполнение работы, без последующего переопределения графика.
4. Сократить этапы работ. На практике это происходит в любом случае, как только команда замечает отставание от графика. Там, где вторичные затраты из-за задержки очень высоки, это действие является единственно возможным. Менеджеру предоставляется возможность официально и аккуратно сократить задачу, изменить график либо наблюдать, как задача молча урезается при поспешном изменении проекта и неполном тестировании.

В первых двух случаях будет катастрофой настаивать на том, чтобы исходный проект был выполнен в течение 4 месяцев. Возьмем регенеративные эффекты, например, для первого варианта (рис. 2.8). Два новых человека, какими бы они ни были компетентными и как бы быстро они ни были рекрутированы, потребуют, чтобы их обучил один из опытных сотрудников. Если это займет месяц, то *3 человеко-месяца будут посвящены работе не по первоначальной оценке*. Более того, работа, первоначально разделенная по трем направлениям, должна быть перераспределена по пяти направлениям; следовательно, часть уже проделанной работы будет потеряна, и тестирование системы должно быть продлено. Таким образом, в конце третьего месяца остается значительно больше 7 человеко-месяцев усилий, и имеются 5 обученных людей и 1 месяц. Как подсказывает рис. 2.8, продукт все равно будет опаздывать, словно никто и не был добавлен (рис. 2.6).

Для того чтобы надеяться, что все будет сделано за 4 месяца, учитывая только время обучения и отсутствие перераспределения работы и дополнительного тестирования системы, в конце второго месяца потребуется добавить четырех человек, а не двух. Для покрытия отрицательных эффектов перераспределения и тести-

рования системы потребовалось бы добавить еще специалистов. Теперь, однако, имеется по крайней мере команда из семи человек, а не из трех; следовательно, такие аспекты, как организация команды и разделение работы, уже различаются качественно, а не только количественно.

Обратите внимание, что к концу третьего месяца все выглядит весьма мрачно. Несмотря на все управленческие усилия, контрольная точка 1 марта не была достигнута. Очень велик соблазн повторить цикл, добавив еще больше рабочей силы. И в этом кроется безумие.

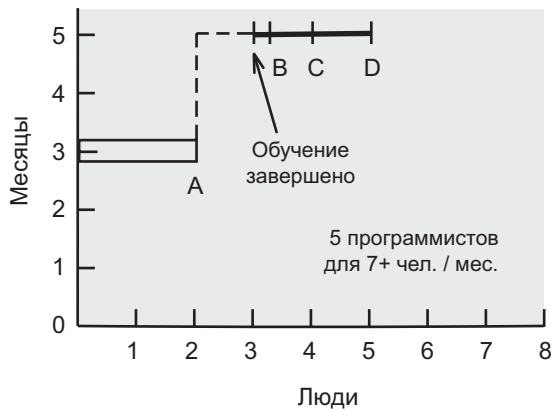


Рис. 2.8

Изложенное выше исходило из того, что только первая контрольная точка была оценена неправильно. Если 1-го марта принять консервативное допущение, что весь график был оптимистичным, как изображено на рис. 2.7, то в исходную работу нужно добавить только 6 человек. Расчет эффектов обучения, перераспределения, системного тестирования оставлен читателю в качестве упражнения. Без сомнения, регенеративная катастрофа даст более слабый продукт и позже, чем дала бы реструктуризация графика работ с первоначальными тремя исполнителями без их усиления.

Чрезмерно упрощая, сформулируем закон Брукса:

Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше.

В этом и состоит демифологизация человеко-месяца. Длительность проекта зависит от его последовательных ограничений. Максимальное число человек зависит от числа независимых задач. Из этих двух величин можно строить графики проекта, используя меньше людей и больше месяцев. (Единственным риском является устаревание продукта.) Вместе с тем нельзя получить работоспособные графики, используя больше людей и меньше месяцев. Случаев, когда проекты по созданию программного обеспечения пошли наперекосяк из-за нехватки календарного времени, гораздо больше, чем неуспешных случаев по всем другим причинам, вместе взятым.



3

ХИРУРГИЧЕСКАЯ БРИГАДА

Эти исследования выявили большие индивидуальные различия между работниками с высокой и низкой производительностью труда, часто на порядок.

Сакман, Эрикссон и Грант¹

На встречах IT-сообщества постоянно можно услышать, как молодые менеджеры утверждают, что отдают предпочтение малой команде первоклассных людей, а не проекту с сотнями программистов, причем явно посредственных. Все мы разделяем это мнение.

Но эта наивная констатация альтернатив позволяет избежать серьезной проблемы — как разрабатывать *крупные* системы по осмысленному графику? Давайте рассмотрим каждую сторону этого вопроса подробнее.

ПРОБЛЕМА

Менеджеры по программированию уже давно признают большие различия в продуктивности между хорошими и плохими программистами. Но фактические измеренные величины поразили всех. В одном из своих исследований Сакман (Sackman), Эриксон (Erikson) и Грант (Grant) измеряли производительность труда в группе опытных программистов. В пределах только этой группы соотношение между лучшими и худшими результатами составляло в среднем около 10:1 по производительности и, что удивительно, 5:1 по скорости и требуемой памяти программы! Словом, программист с годовым окладом \$20 000 может быть в 10 раз продуктивнее, чем программист с окладом \$10 000 в год. Верно и обратное. Данные не показали абсолютно никакой корреляции между опытом и производительностью. (Однако сомневаюсь, что это верно всегда и везде.)

Ранее я аргументировал, что на стоимость работ влияет само число умов, подлежащих координации, так как главенствующая часть затрат связана с коммуникацией и исправлением негативных последствий неправильной коммуникации (отладкой системы). Это также говорит о том, что систему хотят строить как можно меньшим числом людей. Действительно, опыт работы с крупными программными системами показывает, что подход на основе грубой

силы является дорогостоящим, медленным, неэффективным и создает системы, которые не являются концептуально целостными. OS/360, Ehes 8, Score 6600, Multics, TSS, SAGE и др. — этот список будет нескончаемым.

Вывод прост: если проект из 200 человек имеет 25 менеджеров, которые являются наиболее компетентными и опытными программистами, увольте 175 рядовых работников и посадите менеджеров программировать.

Теперь давайте проанализируем это решение. С одной стороны, оно не приближается к идеалу в виде *малой* высококлассной команды, которая, по общему мнению, не должна превышать 10 человек. Она настолько велика, что в ней должно быть не менее двух уровней менеджмента, или около 5 менеджеров. Она также будет нуждаться в поддержке по линии финансов, персонала, пространства, секретарей и операторов машин.

С другой стороны, первоначальной команды из 200 человек недостаточно для того, чтобы создавать действительно крупные системы методами грубой силы. Возьмем, например, OS/360. На пике работ было задействовано более 1000 человек — программисты, составители документации, операторы машин, клерки, секретари, менеджеры, группы поддержки и т. д. С 1963 по 1966 год на ее дизайн, разработку и документирование, вероятно, ушло 5000 человеко-лет. Нашей постулированной команде из 200 человек потребовалось бы 25 лет на то, чтобы довести продукт до его нынешней стадии, если бы люди и месяцы были взаимозаменяемы!

В этом как раз и заключается проблема с концепцией малой высококлассной команды: *она является слишком медленной для действительно крупных систем.* Рассмотрим работу над OS/360, поскольку она может быть решена такой небольшой командой. Обозначим команду из 10 человек. В качестве ограничения пусть они будут в 7 раз продуктивнее посредственных программистов, как

в программировании, так и в документировании, так как являются высококлассными специалистами. Будем считать, что OS/360 создавались только посредственными программистами (что *далеко* от истины). В качестве ограничения допустим, что еще один фактор (множитель) повышения продуктивности, равный семи, возникает из уменьшенной коммуникации со стороны меньшей команды. Будем считать, что выполнять всю работу остается одна и та же команда. Тогда $5000 / (10 \times 7 \times 7) = 10$; они смогут выполнить работу в объеме 5000 человеко-лет за 10 лет. Будет ли продукт интересен через 10 лет после его первоначального дизайна? Или же он устареет в связи с быстро развивающимися программными технологиями?

Данная дилемма является очень болезненной. Ради эффективности и концептуальной целостности предпочтение отдается малому числу хороших умов, занятых дизайном и разработкой. Тем не менее крупные системы должны иметь возможность привлечь солидную рабочую силу с тем, чтобы продукт появился своевременно. Как примирить эти две потребности?

ПРЕДЛОЖЕНИЕ МИЛЛСА

Предложение Харлана Миллса (Harlan Mills) рекомендует свежее и оригинальное решение.^{2,3} Миллс предлагает, чтобы каждый сегмент крупной работы решался командой, но при этом она должна быть организована как хирургическая бригада (команда специалистов), а не как бригада по разделке свиней. То есть вместо того, чтобы каждый член группы орудовал скальпелем на задаче, один делает надрез, а другие оказывают ему всяческую поддержку, которая будет повышать его эффективность и продуктивность.

Небольшое размышление показывает, что эта концепция, если ее заставить работать, соответствует пожеланиям. В дизайне и сбор-

ке участвует мало умов, но задействуется много рук. Будет ли это работать? Кто будет выполнять роли анестезиолога и медсестры в команде по программированию и как работа разделяется между ними? Позволю себе свободное обращение с метафорами, для того чтобы предположить, как такая команда может работать, если ее расширить и включить в нее всю мыслимую поддержку.

ХИРУРГ. Миллс называет его *главным программистом*. Он лично задает спецификации по функциональности и производительности, разрабатывает дизайн программы, кодирует ее, тестирует и пишет документацию. Он пишет на языке структурного программирования, таком как PL/1, и имеет эффективный доступ к вычислительной системе, которая не только выполняет его тесты, но и хранит различные версии его программ, позволяет легко обновлять файлы и обеспечивает редактирование текста в целях документирования. Он имеет большой талант, десятилетний опыт и значительные системные и прикладные знания, будь то прикладная математика, обработка бизнес-данных или что-то еще.

ВТОРОЙ ПИЛОТ. Он является альтер эго хирурга, способен выполнить любую часть работы, но опыта у него меньше. Его главная задача — участвовать в проектировании, где он должен думать, обсуждать, оценивать. Хирург пробует на нем идеи, но не связан по рукам его советами. Второй пилот часто представляет свою команду в обсуждениях с другими командами по поводу функциональности и интерфейса. Он досконально знает весь код. Он исследует альтернативные стратегии дизайна. Очевидно, он служит для хирурга страховкой от катастрофы. Он может даже писать код, но не отвечает за какую-либо его часть.

АДМИНИСТРАТОР. Хирург является начальником, и за ним последнее слово в вопросах персонала, надбавок, пространства и т. д., но он не должен тратить почти все свое время на эти вопросы. Следовательно, ему нужен профессиональный администратор, который управляет деньгами, людьми, пространством и машинами и взаимо-

действует с административным аппаратом остальной организации. Бейкер (Baker) предлагает, чтобы администратор работал полный рабочий день только в том случае, если проект имеет основательные юридические, договорные, отчетные или финансовые требования по причине взаимосвязей между пользователем и производителем. В противном случае один администратор может обслуживать две команды.

РЕДАКТОР. Хирург отвечает за генерирование документации — ради максимальной ясности он должен ее написать. Это справедливо как для внешних, так и для внутренних описаний. Редактор, однако, берет черновик или продиктованную рукопись, подготовленную хирургом, и подвергает ее критическому анализу, перерабатывает, снабжает ссылками и библиографией, пропускает ее через несколько версий и контролирует механику производства.

ДВА СЕКРЕТАРЯ. Администратору и редактору, каждому в отдельности, понадобится секретарь. Секретарь администратора обрабатывает переписку, связанную с проектом, а также документы, не относящиеся к продукту.

ДЕЛОПРОИЗВОДИТЕЛЬ. Он отвечает за регистрацию всех технических данных бригады в библиотеке программного продукта. Он имеет секретарскую подготовку и несет ответственность за все файлы, предназначенные как для машины, так и для чтения.

Весь компьютерный ввод поступает делопроизводителю, который его регистрирует и вводит, если потребуется. Выходные листинги возвращаются к нему для регистрации и индексирования. Самые последние прогоны любой модели хранятся в записной книжке состояния; все предыдущие хранятся в хронологическом архиве.

Жизненно важным для концепции Миллса является преобразование программирования «из приватного ремесла в публичную практику», делая компьютерные программы видимыми для *всех*

членов команды и определяя все программы и данные не как частную собственность, а как командную.

Специализированная функция делопроизводителя освобождает программистов от бюрократических обязанностей, систематизирует и обеспечивает надлежащее выполнение часто забываемых обязанностей, а также увеличивает самый ценный актив команды — ее рабочий продукт. Ясно, что предложенная концепция предполагает прогон пакетных заданий. При использовании интерактивных терминалов, в особенности без вывода на бумажный носитель, функции делопроизводителя не сокращаются, а изменяются. Теперь он регистрирует в журнале все обновления общекомандных копий программы за счет частных рабочих копий, по-прежнему занимается всеми пакетными прогонами и использует свое собственное интерактивное обеспечение для контроля целостности и доступности растущего продукта.

ИНСТРУМЕНТАЛЬЩИК. Благодаря возможности в любое время редактировать файлы и тексты и пользоваться службой интерактивной отладки команде редко требуется своя вычислительная машина и группа обслуживающего персонала. Но доступ к этим службам должен осуществляться с безусловной быстротой и надежностью. Только хирург может решать, удовлетворяет ли его работа имеющихся служб. Ему нужен инструментальщик, ответственный за обеспечение доступа к основным службам, а также за создание, поддержку и обновление специальных инструментов — в основном интерактивных служб, которые требуются его команде. Каждая команда будет нуждаться в своем собственном инструментальщике, независимо от совершенства и надежности любой централизованной службы, поскольку его работа заключается в том, чтобы следить за инструментами, которые необходимы или которые хочет иметь *его* хирург, без учета потребностей любой другой команды. Инструментальщик часто создает специализированные утилиты, каталогизированные процедуры, библиотеки макросов.

ТЕСТИРОВЩИК. Хирургу понадобится банк подходящих тестовых случаев для тестирования частей его работы в процессе написания кода, а затем для тестирования всего результата. Следовательно, тестировщик является одновременно противником, разрабатывающим системные тестовые наборы на основе функциональных спецификаций, и помощником, разрабатывающим тестовые данные для ежедневной отладки. Он также занимается планированием последовательностей тестирования и созданием вспомогательных средств, необходимых для тестирования компонентов.

ЯЗЫКОВОЙ КОНСУЛЬТАНТ. К тому времени, когда появился язык Алгол, люди начали понимать, что в большинстве компьютерных центров всегда найдется один или два человека, которые приходят в восторг от мастерски используемых тонкостей языка программирования. Эти специалисты оказываются очень полезными, и к ним принято обращаться за консультацией. Их талант несколько отличается от таланта хирурга, который в первую очередь является системным дизайнером и мыслит представлениями. Языковой консультант может находить безупречный и эффективный способ использовать язык для того, чтобы делать трудные, смутные или каверзные вещи. Часто ему приходится проводить небольшие исследования (в течение двух-трех дней) по отысканию хорошего технического решения. Один языковой консультант может обслуживать двух или трех хирургов.

Именно так 10 человек могли бы вносить свой вклад, играя хорошо дифференцированные и специализированные роли в команде по программированию, выстроенной на основе хирургической модели.

КАК ЭТО РАБОТАЕТ

Мы определили команду, которая отвечает пожеланиям в нескольких отношениях. Десять человек, семь из них профессионалы,

работают над задачей, но система является продуктом одного-единственного ума — или, самое большее, двух, действующих как единое целое, *uno animo*.

Обратите особое внимание на различия между командой из двух программистов, той, которая организована традиционно, и командой с хирургом и вторым пилотом.

Во-первых, в обычной команде партнеры делят работу, и каждый отвечает за дизайн и имплементацию части работы. В хирургической команде хирург и второй пилот знают весь дизайн и весь код. Это экономит труд по распределению пространства, доступа к диску и т. д. Это также обеспечивает концептуальную целостность работы.

Во-вторых, в обычной команде партнеры равны, и неизбежные различия в суждениях должны проговариваться, или должен найтись компромисс. Поскольку работа и ресурсы поделены, различия в суждениях ограничиваются совокупной стратегией и взаимодействием через интерфейс, но они усугубляются различиями в интересах — например, чье пространство будет использоваться для буфера. В хирургической команде нет различий в интересах, а различия в суждениях разрешаются хирургом в одностороннем порядке. Эти два различия — отсутствие разбиения задачи и отношение подчиненности — позволяют хирургической команде действовать как единое целое.

Также специализация функциональности остальной части команды является ключом к ее эффективности, поскольку она позволяет радикально упростить схему коммуникации между ее членами, как показано на рис. 3.1.

В статье Бейкера сообщается об отдельном маломасштабном тесте указанной командной концепции. Она работала, как и было предсказано для этого случая, с феноменально хорошими результатами.



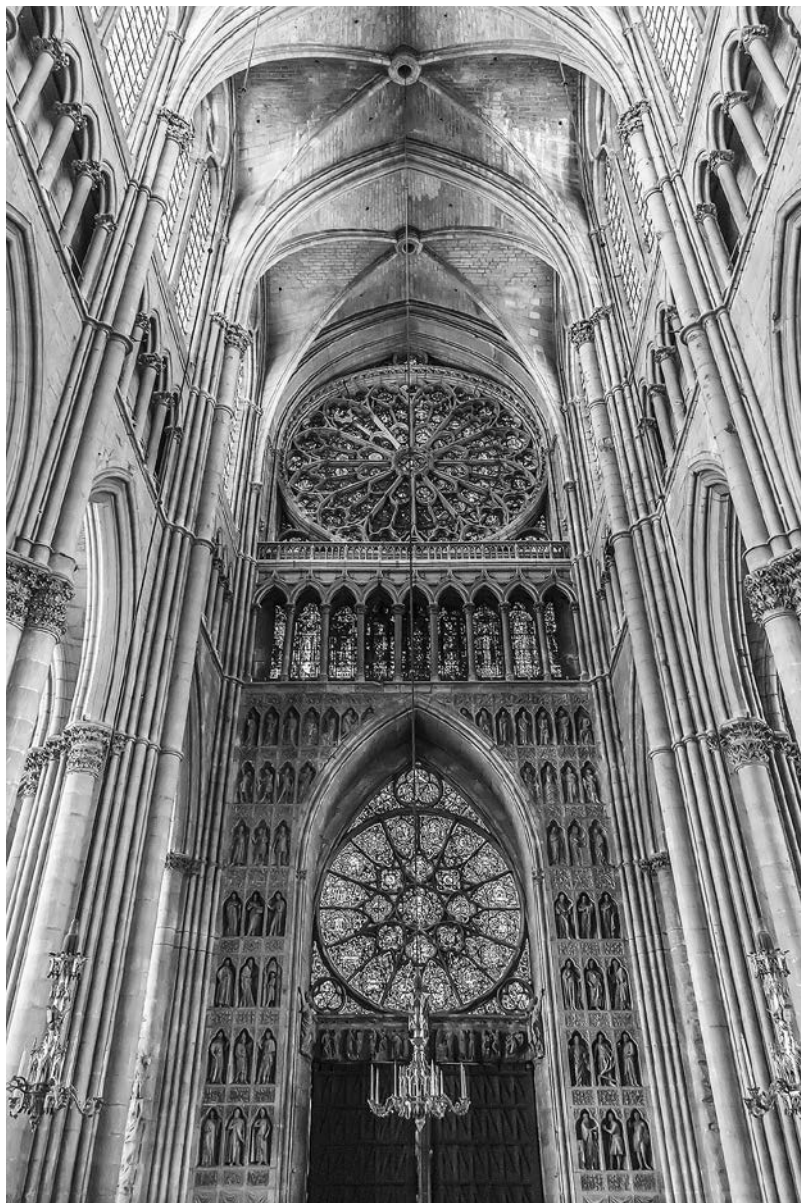
Рис. 3.1. Схемы коммуникации в командах по программированию из 10 человек

ВЕРТИКАЛЬНОЕ МАСШТАБИРОВАНИЕ

Пока все идет хорошо. Проблема, однако, в том, как создавать продукты, которые сегодня занимают 5000 человеко-лет, а не продукты, которые занимают двадцать или тридцать. Команда из 10 человек может быть эффективной независимо от того, как она организована, если вся работа находится в пределах ее компетенции. Но как концепция хирургической команды могла бы использоваться на крупных работах, когда к выполнению задачи привлекается несколько сот человек?

Успех процесса вертикального масштабирования обеспечил и тот факт, что концептуальная целостность каждой части была радикально улучшена — напомним, что число умов, определяющих дизайн, было сокращено в 7 раз. Поэтому можно привлечь к проекту 200 человек, но координировать нужно будет только 20 умов, принадлежащих хирургам.

Однако для решения этой задачи координации необходимо использовать отдельные методики, которые будут рассмотрены в последующих главах. Здесь же достаточно сказать, что вся система также должна иметь концептуальную целостность, и для этого требуется системный архитектор, который будет формировать дизайн всего этого сверху вниз, в нисходящем порядке. Для того чтобы сделать эту работу поддающейся менеджменту, необходимо провести четкое различие между архитектурой и имплементацией, и системный архитектор должен скрупулезно ограничиваться архитектурой. Вместе с тем было показано, что такие роли и методы являются осуществимыми и, по сути, весьма продуктивными.



Фотография Реймского собора. Источник: Википедия

4

АРИСТОКРАТИЯ, ДЕМОКРАТИЯ И СИСТЕМНЫЙ ДИЗАЙН

Эта величественная церковь является выдающимся произведением искусства. В тех догматах, которые она проявляет, нет ни сухости, ни путаницы...

Это зенит стиля, труд художников, которые поняли и усвоили все успехи своих предшественников, в совершенстве владеющих техникой своего времени, но использующих ее без нескромной демонстрации или нелепого проявления умений.

Несомненно, замысел общего плана сооружения принадлежит Жану д'Орбе, который был уважаем, по крайней мере в его существенных элементах, его преемниками. Это одна из причин чрезвычайной согласованности и единства здания.

Путеводитель по Реймскому собору¹

КОНЦЕПТУАЛЬНАЯ ЦЕЛОСТНОСТЬ

Большинство европейских соборов возводились постепенно, и части, построенные строителями разных поколений, различаются в плане архитектурного стиля. Последующие строители испытывали соблазн «совершенствовать» дизайн более ранних, ориентируясь на изменения в архитектурной «моде» и на личный вкус. Поэтому мирный нормандский трансепт* упирается и противоречит парящему готическому нефу, и результат столь же служит восхвалению славы Господней, сколь и гордыни строителей.

На их фоне великолепно контрастирует архитектурное единство Реймса. Восторг, который будоражит зрителя, исходит как от целостности дизайна, так и от любых отдельно взятых достоинств. Как сказано в путеводителе, эта целостность была достигнута самоотречением восьми поколений строителей, каждый из которых пожертвовал некоторыми своими идеями ради чистоты общего замысла. Результат провозглашает не только славу Господню, но и Его могущество, способное спасти грешных людей от их гордыни.

Несмотря на то что для сборки большинства систем программирования не требовались столетия, они демонстрируют гораздо худшую концептуальную разобщенность, чем соборы. Обычно это происходит не из-за смены проектировщиков, а из-за разделения проекта на многочисленные задачи, выполняемые многими людьми.

Я настаиваю на том, что концептуальная целостность является наиболее важным соображением в системном проекте. Лучше иметь систему, в которой отсутствуют некоторые особенности и улучшения, но отражается один набор идей дизайна, чем иметь систему, содержащую много хороших, но независимых и несогласованных

* Поперечный неф в базиликальных и крестообразных по плану храмах, пересекающий основной (продольный) неф под прямым углом. — *Примеч. ред.*

идей. В этой главе и в следующих двух мы рассмотрим последствия этой темы для дизайна систем программирования:

- Как достичь концептуальной целостности?
- Не является ли этот аргумент оправданием для элитарности или аристократизма архитекторов перед ордой плебеев имплементаторов, чьи творческие таланты и идеи подавляются?
- Как удержать архитекторов от дрейфа в голубую даль с неимплементируемыми или дорогостоящими спецификациями?
- Как обеспечить, чтобы каждая незначительная деталь архитектурной спецификации была доведена до сведения имплементатора, правильно им понята и точно встроена в продукт?

ДОСТИЖЕНИЕ КОНЦЕПТУАЛЬНОЙ ЦЕЛОСТНОСТИ

Цель системы программирования — сделать компьютер простым в использовании. Для этого он предоставляет языки и различные средства обеспечения, которые на самом деле являются программами, вызываемыми и управляемыми языковыми возможностями. Но эти средства обеспечения имеют свою цену: внешнее описание системы программирования в 10–20 раз больше, чем внешнее описание самой компьютерной системы. Пользователю гораздо проще задать любую отдельно взятую функцию, но их выбор обширен, и еще гораздо больше вариантов и форматов, о которых нужно помнить.

Использование упрощается только в том случае, если время, выигранное в функциональной спецификации, превышает время, теряемое при усвоении, запоминании и поиске в справочном руководстве. В современных системах программирования этот выигрыш действительно превышает стоимость, но в последние годы отношение выигрыша к стоимости, по-видимому, падало по мере добавления все более и более сложных функций. Меня преследует

воспоминание о простоте использования IBM 650, даже без ассемблера или любого другого программного обеспечения вообще.

Поскольку простота использования является целью, это отношение функциональности к концептуальной целостности является конечной проверкой дизайна системы. Ни функциональность, ни простота сами по себе не определяют хороший дизайн.

Этот тезис широко понимается неправильно. Операционная система OS/360 превозносится ее создателями как самая прекрасная из когда-либо спроектированных, потому что она, бесспорно, имеет самую большую функциональность. Именно функциональность, а не простота всегда была мерой совершенства для ее дизайнеров. С другой стороны, система совместного использования времени для PDP-10 превозносится ее создателями как самая лучшая из-за простоты и сдержанности концепций. Однако по любым меркам ее функциональность даже не относится к тому же классу, что и у OS/360. Как только в качестве критерия принимается простота использования, каждый из этих подходов оказывается несбалансированным, пройдя лишь половину пути до цели.

Однако для заданного уровня функциональности лучше всего подходит та система, в которой можно специфицировать вещи с наибольшей простотой и прямолинейностью. Одной *простоты* недостаточно. Языки TRAC Муерса (Moore) и Algol 68 достигают простоты, измеряемой числом отдельных элементарных понятий. *Непосредственность*, однако, не характерна для них. Чтобы выразить свои намерения, часто требуется сочетать базовые средства сложным и неожиданным образом. Недостаточно изучить элементы и правила их сочетания; нужно также изучить идиоматическое употребление, усвоить целый свод знаний о том, как элементы сочетаются в реальности. Простота и прямолинейность проистекают из концептуальной целостности. Каждая часть должна отражать одну и ту же философию и одно и то же уравнивание пожеланий. Каждая часть должна использовать даже одну и ту же методику в синтаксисе и аналогич-

ные понятия в семантике. Таким образом, простота использования диктует единство дизайна, концептуальную целостность.

АРИСТОКРАТИЯ И ДЕМОКРАТИЯ

Концептуальная целостность, в свою очередь, требует, чтобы проект исходил от одного разработчика или их небольшого числа, действующих согласованно и в унисон.

Давление графика, в свою очередь, требует привлечения большего числа работников. Для решения этой проблемы есть два метода. Первый — это тщательное разделение труда между архитектурой и имплементацией. Второй — это новый способ структурирования команд по имплементации программирования, рассмотренный в предыдущей главе.

Отделение архитектурных усилий от имплементации является мощным способом получения концептуальной целостности в крупных проектах. Я сам видел, как он с большим успехом используется на продуктовой линейке компьютеров IBM Stretch и System/360. И я был свидетелем, как он не сработал при разработке Operating System/360, поскольку недостаточно применялся.

Под *архитектурой* системы я имею в виду полную и детальную спецификацию пользовательского интерфейса. Для компьютера она воплощена в справочном руководстве по программированию. Для компилятора — в справочном руководстве по языку. Для управляющей программы — в справочном руководстве по языку или языкам, используемым для вызова ее функций. Для всей системы она является объединением справочных руководств, помогающих пользователю достичь своей цели.

Архитектор системы, как и архитектор здания, является агентом пользователя. В его обязанности входит использование профес-

сиональных и технических знаний в интересах потребителя, а не в интересах продавца, изготовителя и т. д.²

Архитектура должна четко отличаться от имплементации. Как отметил Бלאув (Blaauw): «Там, где архитектура говорит о том, *что* происходит, имплементация говорит о том, *как* это сделано, чтобы оно произошло».³ В качестве простого примера он приводит часы, архитектура которых состоит из циферблата, стрелок и головки. Когда ребенок усвоит эту архитектуру, он с одинаковой легкостью сможет определять время как по наручным часам, так и по часам на церковной башне. Имплементация же и его реализация описывают, что происходит внутри: передача усилий и управление точностью каждым из многих механизмов.

Например, в System/360 отдельная компьютерная архитектура имплементирована совершенно по-разному в каждой из девяти моделей. В свою очередь, отдельная имплементация, поток данных, память и микрокод системы Model 30 в разное время служит для четырех разных архитектур: компьютера System/360, мультиплексного канала с 224 логически независимыми подканалами, селекторного канала и компьютера 1401.⁴

То же самое различие в равной степени применимо и к системам программирования. В США принят стандарт Fortran IV. Он является архитектурой для многих компиляторов. В рамках этой архитектуры возможны разные реализации: текст в оперативной памяти или компилятор, быстрая или оптимизирующая, синтаксическая или ситуативная (*ad hoc*). Точно так же любой ассемблерный язык или язык управления заданиями допускает многие имплементации ассемблера или планировщика. Теперь мы можем заняться глубоко эмоциональным вопросом аристократии и демократии. Не являются ли архитекторы новой аристократией, интеллектуальной элитой, поставленной для того, чтобы указывать бедным безмозглым имплементаторам, что им делать? Не захватила ли эта элита всю творческую деятельность, сделав исполнителей лишь винтиками

в механизме? Разве нельзя получить более качественный продукт, внедряя хорошие идеи от всей команды, следуя демократической философии, вместо того чтобы ограничивать разработку спецификаций несколькими избранными?

Что касается последнего вопроса, то он — самый простой. Я не утверждаю, что только у архитекторов возникают хорошие архитектурные идеи. Часто новая концепция исходит от имплементатора или от пользователя. Однако весь мой опыт убеждает меня, и я пытался это показать, что концептуальная целостность системы определяет ее простоту использования. Хорошие функции и идеи, которые не интегрируются с базовыми концепциями системы, лучше оставить в стороне. Если таких важных, но несовместимых идей появляется много, то отбрасывается вся система целиком и снова начинается работа над интегрированной системой с другими базовыми концепциями.

Что касается обвинения в аристократизме, то ответ должен быть и «да», и «нет». Да в том смысле, что архитекторов должно быть немного, их продукт должен выстоять дольше, чем продукт имплементатора, и архитектор находится в центре сил, которые он должен, в конечном счете, направить в интересах пользователя. Если система должна обладать концептуальной целостностью, то руководство концепциями должен взять кто-то один. Это аристократизм, который не нуждается в извинениях.

Разработка внешних спецификаций — такая же творческая работа, как и дизайн имплементаций. Это творчество, просто другого рода. Дизайн имплементации с учетом архитектуры требует и допускает столько же дизайнерского творчества, столько же новых идей и столько же технического блеска, сколько дизайн внешних спецификаций. Действительно, соотношение стоимости к производительности продукта в наибольшей степени будет зависеть от имплементатора, так же как простота использования в наибольшей степени зависит от архитектора.

Существует много примеров из области искусств и ремесел, которые заставляют верить, что дисциплина совершенствует мастерство. И действительно, в афоризме художника утверждается: «форма освобождает». Худшие сооружения — это те, чей бюджет был слишком велик для обслуживаемых целей. Творческая деятельность Баха едва ли подавлялась необходимостью еженедельно выпускать ограниченную по форме кантату. Уверен, что компьютер Stretch имел бы более оптимальную архитектуру, если бы он был жестче ограничен; ограничения, налагаемые бюджетом машины System/360 Model 30, по моему мнению, были во всем благотворны для архитектуры Model 75.

Точно так же я замечаю, что внешнее обеспечение архитектуры усиливает, а не сводит на нет творческий стиль группы по имплементированию. Они сразу сосредотачиваются на той части поставленной задачи, которую никто не решал, и изобретения начинают течь рекой. В неограниченной группе по имплементированию большинство мыслей и дискуссий уходит в архитектурные решения, а на имплементацию отводят короткие сроки.⁵

Этот эффект, который я видел много раз, подтверждается Р. В. Конвеем (R. W. Conway), чья группа в Корнелле построила компилятор PL/C для языка PL/1. Он отмечает следующее: «В итоге мы решили реализовать язык без изменений и усовершенствований, поскольку обсуждение языка отняло бы у нас все силы».⁶

ЧЕМ ЗАНЯТЬСЯ ИМПЛЕМЕНТАТОРУ ВО ВРЕМЯ ОЖИДАНИЯ?

Унизительно совершить ошибку стоимостью в миллион долларов, однако так она надолго запоминается. Я отчетливо помню ту ночь, когда мы решили, как организовать фактическое написание внешних спецификаций для OS/360. Мы с менеджером по архитектуре и менеджером по реализации управляющей программы отработали план, график работ и распределение обязанностей.

У менеджера по архитектуре было 10 талантливых людей. Он утверждал, что они могут написать спецификации и сделать это правильно. Это займет 10 месяцев, на три больше, чем позволяет график.

У менеджера по реализации управляющей программы было 150 человек. Он утверждал, что они могли бы подготовить спецификации при координации с архитектурной командой; это было бы сделано хорошо и практично, и он вписался бы в график. Кроме того, если бы этим занялась команда по архитектуре, то его 150 человек сидели бы сложа руки в течение 10 месяцев.

На это менеджер по архитектуре ответил, что если я передам ответственность команде по управляющей программе, то на самом деле результат будет получен на 3 месяца *позже* и гораздо более низкого качества. Я передал ответственность команде по имплементации управляющей программы, и вышло так, как он сказал. Он оказался прав в обоих случаях. Более того, отсутствие концептуальной целостности сделало систему гораздо более дорогостоящей при сборке и изменении, и, по моим оценкам, это на год задержало отладку.

Разумеется, на это ошибочное решение повлияли многие факторы; но подавляющими из них были временные ограничения графика и апелляция к привлечению всех этих 150 имплементаторов к работе. Именно это пение сирен, таящих смертельные опасности, я и сделаю сейчас видимым.

На предложение, что небольшая команда по архитектуре фактически напишет все внешние спецификации для компьютера или системы программирования, имплементаторы выдвигают три возражения:

- Спецификации будут слишком насыщены функциональностью и не будут отражать практическую стоимость.
- Архитекторы получают все творческое удовольствие и откажутся от изобретательности имплементаторов.

- Многочисленным исполнителям придется сидеть сложа руки, пока спецификации пройдут через узкое горлышко команды архитекторов.

Первое из них представляет реальную опасность, и мы рассмотрим ее в следующей главе. Две других — это иллюзии, чистые и простые. Как мы увидели выше, имплементация тоже является творческой деятельностью первого порядка. Возможность проявить творчество и изобретательность при разработке незначительно ограничивается необходимостью работать в рамках заданных внешних спецификаций, и такая дисциплина может даже усилить степень творчества. Это, несомненно, верно для проекта в целом.

Последнее возражение касается сроков и фаз. Быстрый ответ — не нанимать имплементаторов до тех пор, пока спецификации не будут завершены. В строительстве действуют по тому же принципу.

В бизнесе компьютерных систем, однако, темпы оказываются быстрее, и каждый хочет сжать график как можно больше. В какой степени спецификация и разработка могут накладываться друг на друга?

Как указывает Блааув, общее творческое усилие включает в себя три разные фазы: архитектуру, имплементацию и реализацию. Оказывается, что они действительно могут быть начаты параллельно и продолжаться одновременно.

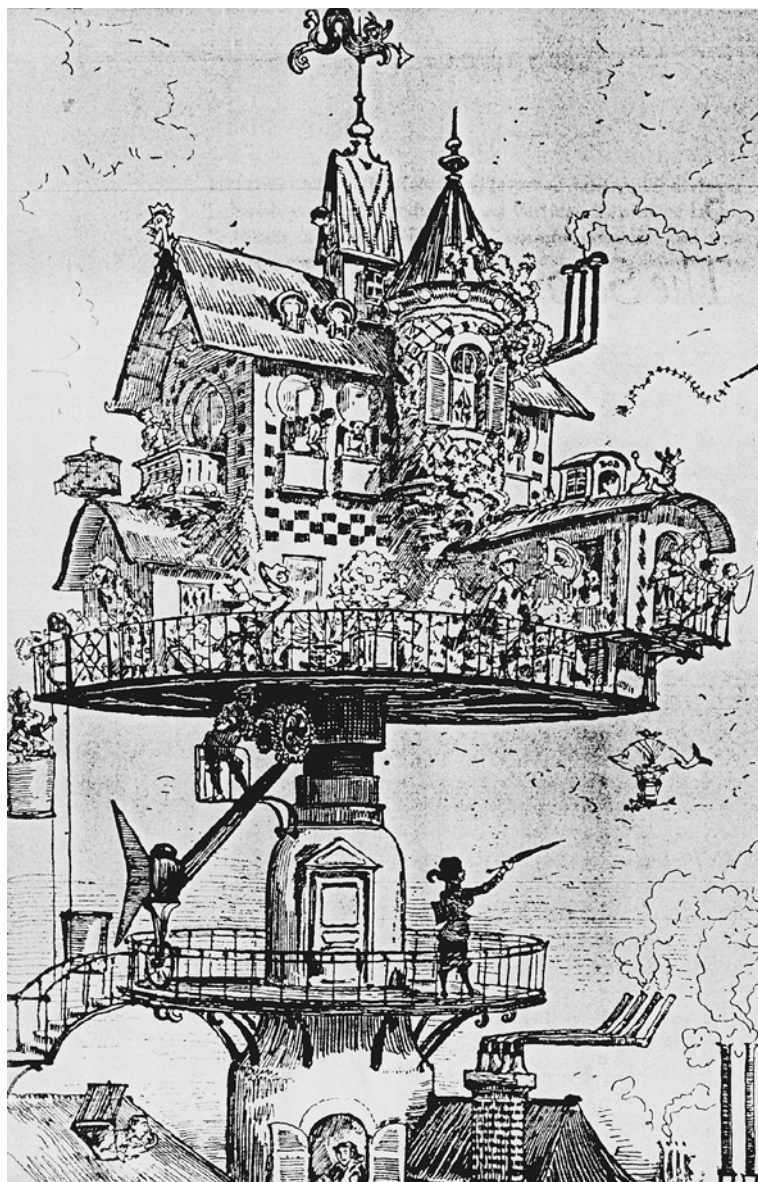
Например, в дизайне компьютеров имплементатор может начать работу, как только у него появятся относительно расплывчатые допущения о справочном руководстве, более-менее четкие идеи о технологии и четко определенные целевые критерии по стоимости и производительности. Он может заняться дизайном потоков данных, управляющих последовательностей, грубых концепций упаковки и т. д. Он разрабатывает или адаптирует инструменты, которые ему понадобятся, в особенности систему учета, включая систему автоматизации дизайна.

В то же время на уровне реализации должен быть составлен, усовершенствован и задокументирован дизайн схем, карт, кабелей, рамок, источников питания и памяти. Эта работа идет параллельно с архитектурой и имплементацией.

То же верно и для дизайна систем программирования. Задолго до того, как внешние спецификации будут завершены, у имплементатора есть много дел. С учетом некоторых грубых упрощений касательно функциональности системы, которая в конечном счете будет воплощена во внешних спецификациях, он может продолжить работу. У него должны быть четко определенные пространственные и временные целевые критерии. Он должен знать конфигурацию системы, на которой должен работать его продукт. Затем он может приступить к дизайну границ модулей, структур таблиц, разбиений на проходы и фазы, алгоритмов и всевозможных инструментов. Некоторое время также должно быть потрачено на коммуникацию с архитектором.

На уровне реализации еще много работы. Программирование тоже имеет технологию. Если машина — новая, предстоит сделать многое относительно соглашений о вызове подпрограмм, технологии работы с супервизором, алгоритмов поиска и сортировки.⁷

Концептуальная целостность требует, чтобы система отражала единую философию и чтобы видимую пользователю спецификацию писали несколько человек. Однако вследствие реального разделения труда на архитектуру, имплементацию и реализацию из этого вовсе не следует, что сборка спланированной таким образом системы займет больше времени. Опыт показывает обратное: что целостная система сходится все быстрее и быстрее и требует меньше времени на тестирование. По сути дела, широко распространенное горизонтальное разделение труда было резко сокращено вертикальным разделением труда, и результатом этого стало радикальное упрощение коммуникаций и улучшение концептуальной целостности.



Поворотная вышка для воздушного движения.
Литография, Париж, 1882 год. Из *Vintieme Cicle*, A. Robida

5

ЭФФЕКТ ВТОРОЙ СИСТЕМЫ

Adde parvum parvo magnus acervus erit.

[Добавляй малое к малому, и ты получишь большую кучу.]

Овидий

Если разделить ответственность за функциональную спецификацию от ответственности за разработку быстрого и дешевого продукта, то какая дисциплина ограничивает изобретательский энтузиазм архитектора?

Фундаментальный ответ заключается в основательной, тщательной и симметричной коммуникации между архитектором и разработчиком. Тем не менее есть более тонкие ответы, которые заслуживают внимания.

ИНТЕРАКТИВНАЯ ДИСЦИПЛИНА ДЛЯ АРХИТЕКТОРА

Архитектор в строительстве работает на основании бюджета, используя методы оценивания, которые позже подтверждаются или корректируются заявками подрядчиков. Часто бывает так, что все заявки превышают бюджет. Тогда на следующей итерации архитектор пересматривает свою методику оценивания в сторону повышения, а свой дизайн — в сторону понижения. Он, возможно, предложит подрядчикам более дешевый способ имплементации его дизайна, чем они разработали.

Аналогичный процесс управляет архитектором компьютерной системы или системы программирования. Однако у него есть то преимущество, что предложения подрядчика можно получить на ранних стадиях проектирования, часто — в любой момент. Но обратная сторона медали в том, что он работает только с одним подрядчиком, который может поднять или понизить свои оценки и бюджета, и сроков, и ресурсов, в зависимости от того, насколько он доволен результатом. На практике ранняя и непрерывная коммуникация может дать архитектору хорошие показатели стоимости, а разработчику уверенность в дизайне без размывания четкого разделения обязанностей.

У архитектора есть два возможных ответа, когда он сталкивается с оценкой, являющейся слишком высокой: урезать дизайн или

бросить вызов оценке, предложив более дешевые имплементации. Последнее, в сущности, является деятельностью, генерирующей эмоции. Архитектор теперь бросает вызов тому, как строитель выполняет свою работу. Для того чтобы она стала успешной, архитектор должен:

- помнить, что разработчик несет изобретательную и творческую ответственность за имплементацию; поэтому архитектор предлагает, а не диктует;
- всегда быть готовым предложить *способ* имплементации всего, что он специфицирует, а также быть готовым принять любой другой способ, который отвечает целевым критериям;
- вести дела спокойно и действовать без огласки в таких предложениях;
- не рассчитывать на признательность за сделанные предложения.

Обычно разработчик будет противостоять, предлагая изменения в архитектуре. Часто он прав — какое-то незначительное свойство может повлечь неожиданно большие затраты при разработке имплементации.

САМОДИСЦИПЛИНА — ЭФФЕКТ ВТОРОЙ СИСТЕМЫ

Первая работа архитектора, как правило, является щадящей и чистой. Он знает, что не знает что делает, поэтому делает это осторожно и очень сдержанно.

По мере того как он работает над дизайном первой работы, ему на ум приходят украшение за украшением, фишка за фишкой. Они будут сохранены для использования «в следующий раз». Рано или поздно первая система будет закончена, и архитектор, с твердостью уверенностью и демонстративным мастерством владения этим классом систем, готов построить вторую систему.

Эта вторая система — самая опасная для человека, который ее проектирует. Когда он трудится над своей третьей и более поздними, все экземпляры из его предшествующего опыта будут подтверждать друг друга относительно общих характеристик таких систем, и их различия будут определять те части его опыта, которые являются частными и необобщаемыми.

Общая тенденция — делать дизайн второй системы с большим запасом, используя все идеи и излишества, которые были осторожно отклонены в первой. В результате, как говорил Овидий, получается «большая куча». Например, возьмем архитектуру IBM 709, позже воплощенную в 7090. Вторая система является модернизацией очень успешной и чистой архитектуры 704. Набор команд был настолько богат и насыщен, что регулярно использовалась примерно лишь половина.

В качестве более яркого примера возьмем архитектуру, имплементацию и даже реализацию компьютера Stretch, являющегося выходом для сдерживаемых изобретательских желаний многих людей и второй системой для большинства из них. Как говорит Стрейчи (Strachey) в своем обзоре:

«У меня сложилось впечатление, что Stretch — это в некотором роде окончание определенного направления разработок. Как и некоторые ранние компьютерные программы, указанная система является чрезвычайно изобретательной, чрезвычайно сложной и эффективной, но в то же время сырой, расточительной и незлегантной, и каждый чувствует, что должен быть некий оптимальный способ того, как делаются вещи».¹

Операционная система OS/360 была второй системой для большинства ее дизайнеров. Группы ее дизайнеров пришли из разработки дисковой операционной системы 1410-7010, операционной системы Stretch, системы реального времени Project Mercury и IBSYS для 7090. Вряд ли кто-то имел опыт работы с *двумя*

предыдущими операционными системами.² И поэтому OS/360 является ярким примером эффекта второй системы, отрезком* программного ремесла, к которому относятся как похвалы, так и упреки критики Стрейчи.

Например, OS/360 отводит 26 байт резидентной процедуре смены дат для правильной обработки 31 декабря в високосные годы (когда 366 дней в году). Это можно было бы оставить оператору.

Эффект второй системы имеет еще одно проявление, несколько отличное от чисто функционального приукрашивания. Это тенденция к совершенствованию методики, само существование которой было деактуализировано изменениями в базовых допущениях системы. OS/360 имеет много тому примеров.

Возьмем редактор связей, предназначенный для загрузки отдельно скомпилированных программ и урегулирования их перекрестных ссылок. Помимо выполнения этой основной функции, он также обрабатывает оверлеи (программные наложения). Он является одним из лучших когда-либо построенных оверлейных средств обеспечения. Позволяет структурировать оверлей внешне, во время компоновки, не будучи конструктивно встроенным в исходный код. Позволяет изменять структуру оверлея от прогона к прогону без перекомпиляции. Предоставляет насыщенное разнообразие полезных опций и средств. В каком-то смысле он является кульминацией многолетнего развития статического оверлейного метода.

Но это также последний и лучший из динозавров, поскольку он принадлежит системе, в которой мультипрограммирование является нормальным режимом, а динамическое распределение ядра — базовым допущением. Это находится в прямом противоречии

* Здесь автор приводит каламбур, используя слово Stretch с заглавной буквы, которое является и торговой маркой компьютера, и переводится как «отрезок», то есть является окончанием отдельно взятой линии развития. — *Примеч. пер.*

с пониманием использования статических оверлеев. Насколько лучше работала бы система, если бы усилия, направленные на управление оверлеями, были потрачены на то, чтобы сделать средства обеспечения динамического распределения ядра и динамического урегулирования перекрестных ссылок действительно быстрыми!

Более того, редактор связей требует так много места и сам содержит так много оверлеев, что даже когда он используется только для компоновки без управления оверлеем, он работает медленнее, чем большинство системных компиляторов. Ирония состоит в том, что назначение редактора связей — избежать повторной компиляции. Как у конькобежца корпус оказывается впереди ног, так и усовершенствования продолжались, пока не вышли далеко за рамки системных принципов.

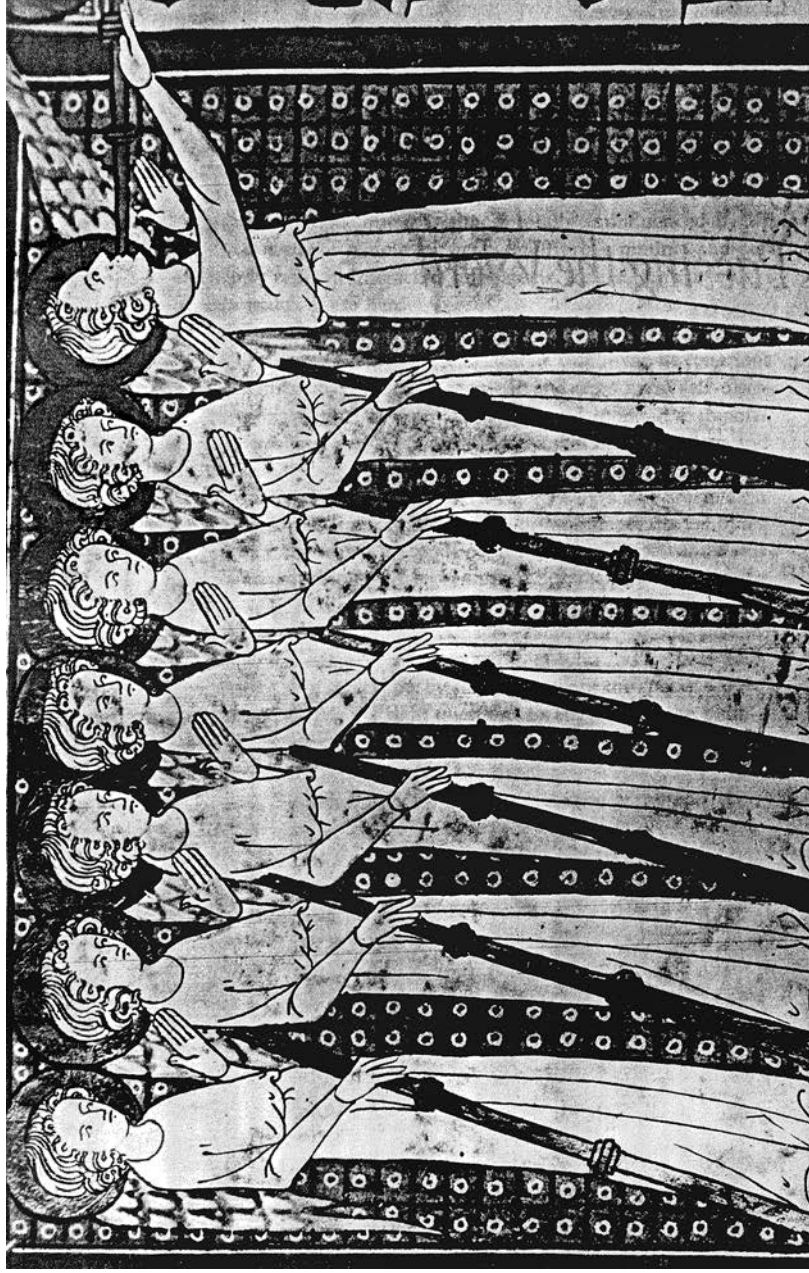
Отладочное средство обеспечения TESTRAN является еще одним примером этой тенденции. Это совершенный пакетный отладчик, предоставляющий действительно элегантные средства получения мгновенных снимков и дампов памяти. В нем используется концепция управляющей секции и гениальный генераторный метод, допускающий выборочную трассировку и привязку без накладных расходов на интерпретацию или без перекомпиляции. Творческие концепции операционной системы Share Operating System³ для 709 расцвели здесь пышным цветом.

Тем временем сама идея пакетной отладки без повторной компиляции стала устаревать. Интерактивные вычислительные системы, использующие языковые интерпретаторы или инкрементные компиляторы, бросили наиболее фундаментальный вызов. Но даже в пакетных системах появление компиляторов быстрой компиляции/медленного исполнения сделало отладку и привязку на уровне исходного кода предпочтительным методом. Насколько лучше оказалась бы система, если бы силы, потраченные на проект TESTRAN, были направлены на ускоренное создание лучших средств для интерактивной работы и быстрой компиляции!

Еще одним примером является планировщик, который предоставляет поистине превосходные средства для управления потоком заданий в режиме фиксированных пакетов. В практическом смысле этот планировщик является усовершенствованной, улучшенной и приукрашенной второй системой после дисковой операционной системы 1410-7010, пакетной системой без мультипрограммирования, за исключением ввода-вывода и предназначенной главным образом для бизнес-приложений. Как таковой, планировщик OS/360 является хорошим. Но на него существенно повлияли потребности OS/360 в удаленном вводе заданий, в мультипрограммировании и резидентных интерактивных подсистемах. В действительности дизайн планировщика затрудняет их.

Как архитектор избегает эффекта второй системы? Совершенно очевидно, что он не может пропустить свою вторую систему. Но он может осознавать, какие особые опасности его подстерегают, и проявлять дополнительную самодисциплину во избежание функциональных приукрашиваний и экстраполяции функций, которые стали несостоятельными в связи с изменениями в допущениях и целях.

Дисциплина, которая откроет глаза архитектору, заключается в том, чтобы назначить каждой маленькой функции значение: способность x сто́ит не более m байт памяти и n микросекунд на вызов. Эти значения будут определять первоначальные решения и служить в ходе имплементации в качестве руководящего принципа и предупреждения для всех. Как менеджер проекта избегает эффекта второй системы? Настаивать на том, чтобы старший архитектор имел опыт разработки хотя бы двух систем. Кроме того, будучи осведомленным о возможных опасностях, он может предъявлять необходимые требования для того, чтобы в подробном проекте нашли полное отражение идеологические концепции и цели.



Семь труб Апокалипсиса, XIV век. Архив Бетмана

6

ДОВЕДЕНИЕ ДО СВЕДЕНИЯ

Он сядет здесь и будет распоряжаться:
«Сделайте это! Сделайте то!» А дело
и с места не сдвинется.

Гарри Трумэн. О президентской власти¹

Исходя из того, что у менеджера есть дисциплинированные, опытные архитекторы и что в проекте задействовано большое количество имплементаторов, каким образом он достигает того, чтобы каждый из них слышал, понимал и имплементировал решения архитекторов? Каким образом группа из 10 архитекторов поддерживает концептуальную целостность системы, которую разрабатывает тысяча человек? Для этого существует целая технология, которая была отработана на дизайне аппаратного обеспечения System/360, и она в равной степени применима к проектам по сборке ПО.

ПИСЬМЕННЫЕ СПЕЦИФИКАЦИИ — СПРАВОЧНОЕ РУКОВОДСТВО

Справочное руководство (мануал), или письменная спецификация, является необходимым инструментом, хотя и недостаточным. Справочное руководство представляет собой *внешнюю* спецификацию продукта. В ней описывается и прописывается каждая деталь того, что пользователь видит. Как таковое, оно является главным продуктом архитектора.

Круг за кругом цикл ее подготовки продолжается по мере того, как обратная связь от пользователей и имплементаторов показывает места, где дизайн неудобен в использовании или разработке. Для удобства необходимо квантовать изменения: согласно определенным в графике датам выпускать очередные версии.

Справочное руководство не только должно описывать все, что видит пользователь, включая все интерфейсы, но и воздерживаться от описания того, что пользователь не видит. Это дело имплементатора, и там его свобода должна быть неограниченной. Архитектор всегда должен быть готов показать *некую* имплементацию любой *функциональной особенности*, которую он описывает, но он не должен пытаться диктовать *конкретную* имплементацию.

Стиль должен быть ясным, полным и точно детализированным. Пользователь часто ссылается на одно-единственное определение, поэтому его упоминание должно повторять все существенные части, и при этом все упоминания должны быть в согласии. В результате этого чтение справочного руководства, как правило, становится скучным занятием, но в данном случае важнее точность, чем живость изложения.

Единство «Принципов работы» системы System/360 проистекает из того, что принадлежит перу только двух человек: Джерри Блааува и Андреса Падегса. Идеи принадлежат примерно 10 людям, но воплощение этих решений в текстовые спецификации должно быть проделано только одним человеком или двумя, если нужно сохранить единообразие и согласованность описания и продукта. Ибо написание определения потребует целого ряда мини-решений, которые не имеют дискуссионной важности. Примером в System/360 является подробное описание того, как устанавливается код состояния после каждой операции. Нетривиальным, однако, является принцип, что такие мини-решения должны приниматься единообразно во всем.

Думаю, самым прекрасным элементом написания справочного руководства, который я когда-либо видел, является приложение Блааува к «Принципам работы» System/360. Автор с осторожностью и точностью очертил пределы совместимости System/360. В руководстве дается определение совместимости, прописывается то, что должно быть достигнуто, и перечисляются те области внешнего вида, где архитектура намеренно молчит и где результаты одной модели могут отличаться от результатов другой, где одна копия данной модели может отличаться от другой копии или где копия может отличаться даже от самой себя после изменения. Это тот уровень точности, к которому стремятся авторы справочных руководств, и они должны определять то, что не прописано так же тщательно, как то, что прописано.

ФОРМАЛЬНЫЕ ОПРЕДЕЛЕНИЯ

Английский или любой другой человеческий язык, естественно, не является точным инструментом для таких определений. Поэтому составитель справочного руководства должен потрудиться ради достижения необходимой точности. Привлекательной альтернативой является использование формальных обозначений. В конце концов точность является *смыслом существования* формальных обозначений.

Давайте рассмотрим достоинства и недостатки формальных определений. Как уже отмечалось, формальные определения являются точными. Они имеют тенденцию быть полными; пробелы проявляются заметнее, поэтому заполняются быстрее. Чего им не хватает, так это ясности. Их недостаток — трудность понимания. На английском языке можно описать структурные принципы, очертить структуры по этапам или по уровням и привести примеры. Можно легко отметить исключения и подчеркнуть контрасты. Самое главное, можно объяснить *причину*. Предлагавшиеся до сих пор формальные определения вызывали восхищение своей элегантностью и кажущейся точностью. Но они требовали текстуальных пояснений для облегчения изучения своего содержания. По этой причине я полагаю, что в будущем спецификации будут состоять как из *формальных*, так и из *текстовых* описаний.

Древняя пословица предупреждает: «Никогда не выходи в море с двумя хронометрами; возьми один или три». То же самое явно относится к текстовым и формальным определениям. Если есть и то и другое, то одно должно быть стандартным описанием, а другое — производным, четко обозначенным как таковое. Любое из них может быть первичным стандартом. Algol 68 имеет формальное определение в качестве стандартного и текстовое определение в качестве описательного. PL/1 имеет текстовое определение в качестве стандартного и формальное описание в качестве производ-

ного. System/360 также имеет текстовое определение в качестве стандартного с производным формальным описанием.

Для формального определения имеется ряд инструментов. Форма Бэкуса-Наура широко используется для определения языков и подробно обсуждается в литературе.² В формальном описании языка PL/1 используются новые понятия абстрактного синтаксиса, и он описан соответствующим образом.³ APL Айверсона (Iverson) был использован для описания машин, в частности IBM 7090⁴ и System/360.⁵

Белл (Bell) и Ньюэлл (Newell) предложили новые обозначения для описания как конфигураций, так и машинных архитектур, и они проиллюстрировали их несколькими машинами, включая DEC PDP-8⁶, 7090⁶ и System/360⁷.

Почти все формальные определения оказываются воплощением или описанием имплементации аппаратной или программной системы, внешние характеристики которой они прописывают. Синтаксис может быть описан без этого, но семантика обычно определяется путем предоставления программы, которая выполняет определенную операцию. Это, конечно же, является имплементацией и как таковое чрезмерно прописывает архитектуру. Поэтому нужно позаботиться о том, чтобы указать, что формальное определение применимо только к внешним проявлениям, и уточнить, какими они бывают.

Не только формальное определение является имплементацией, но и имплементация может служить формальным определением. Когда были построены первые совместимые компьютеры, использовался именно этот метод. Новая машина должна была соответствовать существующей машине. Справочное руководство было расплывчатым по некоторым пунктам — «Спроси у машины!» Разрабатывалась тестовая программа, которая служила для определения поведения, и новая машина строилась в полном соответствии.

Программируемый эмулятор аппаратной или программной системы может служить точно таким же образом. Он является имплементацией, он работает, и поэтому все вопросы об определении могут быть решены путем его тестирования.

Использование имплементации в качестве определения имеет свои преимущества. Все вопросы могут однозначно быть решены путем эксперимента. Обсуждений не требуется, поэтому ответы даются быстро. Ответы всегда имеют ту точность, которой хочется, и всегда по определению являются правильными. С другой стороны, такой подход имеет существенные недостатки. Имплементация может чрезмерно прописывать даже внешние факторы. Недопустимый синтаксис всегда приводит к некоторому результату; в контролируемой системе этот результат является признаком недопустимости, *и ничем больше*. В не доведенной до блеска системе могут появиться всевозможные побочные эффекты, и они, возможно, были использованы программистами. Когда мы взялись эмулировать IBM 1401 на System/360, выяснилось, что существует 30 различных «курьезов» — побочных эффектов якобы недопустимых операций, которые получили широкое распространение и должны были рассматриваться как часть определения. Имплементация в качестве определения была прописана чрезмерно; в ней не только указывалось, что машина должна делать, но и подробно говорилось о том, как она должна это делать.

Имплементация также иногда дает неожиданные и незапланированные ответы, когда задаются острые вопросы, и определение *de facto* часто оказывается неэлегантным в этих деталях именно потому, что оно никогда не было продумано. Такая неэлегантность часто ведет к тому, что ее дублирование в другой имплементации оказывается медленным или дорогим. Например, некоторые машины оставляют мусор в регистре множимого после умножения. Точная природа этого мусора оказывается частью определения *de facto*, но его дублирование может препятствовать использованию более быстрого алгоритма умножения.

Наконец, использование реализации в качестве формального определения может создать неясность, какое из описаний — текстовое или формальное — в действительности является стандартом. Это в особенности верно для программного моделирования. Необходимо также воздерживаться от внесения модификаций в имплементацию в то время, когда она служит стандартом.

ПРЯМОЕ ВСТРАИВАНИЕ

У архитектора программной системы есть прекрасный метод пространства и обеспечения определений. Он полезен для установления синтаксиса, если вообще не семантики, межмодульных интерфейсов. Этот метод заключается в дизайне объявления передаваемых параметров или совместного хранилища и требует, чтобы имплементации включали это объявление посредством операции времени компиляции (макрос или директива `%INCLUDE` в PL/1). Если, кроме того, все ссылки на интерфейс происходят только по символическим именам, объявления можно менять, добавляя или вставляя новые имена и лишь заново компилируя, но не изменяя использующую его программу.

КОНФЕРЕНЦИИ И «СУДЫ»

Излишне говорить, что совещания необходимы. Сотни личных встреч должны быть дополнены более крупными и официальными собраниями. Мы нашли полезными два уровня из них. Первый — это еженедельная полудневная конференция всех архитекторов и официальных представителей имплементаторов аппаратного и программного обеспечения, а также представителей бизнеса. Встречу ведет главный системный архитектор.

Любой желающий может предложить задачи или изменения, но предложения обычно распространяются в письменном виде до

начала совещания. Новая задача обсуждается некоторое время. Акцент делается на творчестве, а не просто на решении. Группа пытается изобрести целый ряд решений задач, затем несколько решений передаются одному или нескольким архитекторам для детализации в точно сформулированные предложения по изменению справочного руководства.

Затем подробные предложения по изменениям поступают для принятия решений. Они были предварительно розданы и тщательно рассмотрены имплементаторами и пользователями, а их плюсы и минусы четко очерчены. Если возникнет консенсус, то это только во благо. Если нет, то решение принимает главный архитектор. Ведется протокол, и решения формально, оперативно и широко обсуждаются.

Еженедельные конференции дают быстрые результаты и позволяют продолжить работу. Если кто-то *категорически* недоволен решением, то возможны немедленные обращения к менеджеру проекта, однако это случается очень редко.

Плодотворность таких совещаний проистекает из нескольких источников:

1. Одна и та же группа — архитекторы, пользователи и имплементаторы — собирается еженедельно в течение нескольких месяцев. Не требуется времени на то, чтобы вводить людей в курс дела.
2. Яркая, находчивая группа хорошо разбирается в вопросах и глубоко вовлечена в результат. Ни у кого нет «консультативной» роли. Каждый уполномочен принимать на себя твердые обязательства.
3. Когда поднимаются проблемы, решения отыскиваются как внутри, так и за пределами очевидных границ.
4. Формальность письменных предложений фокусирует внимание, заставляет принимать решения и позволяет избегать противоречий в черновике, разработанном комитетом.

5. Четкое наделение главного архитектора полномочиями по принятию решений позволяет избегать компромиссов и задержек.

С течением времени некоторые решения себя не оправдывают. По некоторым незначительным вопросам невозможно прийти ко всеобщему согласию. Другие решения порождают непредвиденные сложности, но иногда еженедельное совещание не соглашается их пересматривать. В результате накапливается ряд мелких апелляций, открытых вопросов или недовольств. Чтобы закрыть этот ряд проблем, мы проводили ежегодные «заседания верховного суда», длящиеся обычно две недели. (Я бы проводил их каждые шесть месяцев, если бы снова этим занимался.)

Эти сессии проводились непосредственно перед решающими датами «заморозки» справочного руководства. В них принимали участие не только группа архитекторов, представители программистов и имплементаторов, но и менеджеры по программированию, маркетингу и имплементации. Председательствовал менеджер проекта System/360. Повестка работы включала обычно около 200 пунктов, в основном мелких, перечисленных в развешанных по комнате списках. Заслушивались все стороны и принимались решения. Благодаря чуду компьютерной верстки (и превосходной работе сотрудников) каждое утро каждый участник обнаруживал на своем рабочем месте исправленное руководство, в которое были внесены решения, принятые накануне. Эти «осенние фестивали» оказались полезны не только для решения проблем, но и для их признания. Все были услышаны, все участвовали, все лучше понимали сложные ограничения и взаимосвязи между решениями.

МНОГОЧИСЛЕННЫЕ ИМПЛЕМЕНТАЦИИ

Архитекторы System/360 имели два почти беспрецедентных преимущества: достаточно времени для тщательной работы и политическое влияние, равное влиянию имплементаторов. Обеспечение

достаточного количества времени вытекало из графика новой технологии; политическое равенство вытекало из одновременной разработки многочисленных имплементаций. Необходимость строгой совместимости между ними служила наилучшим возможным средством обеспечения соблюдения спецификаций.

В большинстве компьютерных проектов наступает день, когда обнаруживается, что машина и справочное руководство не согласуются. В этом противостоянии справочное руководство обычно проигрывает, поскольку его можно изменить гораздо быстрее и дешевле, чем машину. Однако это не так, когда существует несколько имплементаций. Тогда задержки и затраты, ассоциированные с исправлением ошибочной машины, могут быть компенсированы задержками и затратами на ревизию машин, которые верно следовали справочному руководству.

Эта идея может плодотворно применяться всякий раз, когда определяется язык программирования. Можно быть уверенным, что рано или поздно будет построено несколько интерпретаторов или компиляторов для достижения различных целевых критериев. Определение будет чище, а дисциплина жестче, если изначально будут созданы по крайней мере две имплементации.

ТЕЛЕФОННЫЙ ЖУРНАЛ

В ходе имплементации возникают бесчисленные вопросы по архитектуре, независимо от того, насколько точной является спецификация. Очевидно, что такие моменты требуют усиления и разъяснения в тексте. Другие просто свидетельствуют о недопонимании.

Крайне важно, однако, побудить озадаченного имплементатора позвонить ответственному архитектору и задать свой вопрос, а не в раздумьях продолжать работу. Столь же важно признать, что ответы на такие вопросы представляют собой архитектурные объ-

явления со своего рода *университетской кафедры*, которые должны быть доведены до каждого.

Одним из полезных механизмов является *телефонный журнал*, который ведет архитектор. В нем он записывает каждый вопрос и каждый ответ. Каждую неделю журналы нескольких архитекторов объединяются, размножаются и распространяются среди пользователей и имплементаторов. Хотя этот механизм довольно неформальный, он является быстрым и всеобъемлющим.

ТЕСТ ПРОДУКТА

Лучший друг менеджера проекта — его ежедневный противник, независимая организация по тестированию продукта. Эта группа проверяет машины и программы в соответствии со спецификациями и служит адвокатом дьявола, выявляя все мыслимые ошибки и несоответствия. Для того чтобы оставаться честным, каждый коллектив разработчиков нуждается в такой независимой технической аудиторской группе.

В конечном счете заказчик является независимым аудитором. В безжалостном свете реального использования проявится каждый изъян. Группа тестирования продукта является суррогатным клиентом, специализирующимся на выявлении недостатков. Раз за разом дотошный тестировщик продукта будет находить места, где информация не была передана, где решения по дизайну не были правильно поняты или точно имплементированы. По этой причине группа тестирования является необходимым звеном в цепи, по которой передается информация о дизайне, звеном, которое должно действовать на ранней стадии и одновременно с дизайном.



Питер Брейгель Старший. «Вавилонская башня», 1563. Источник: Википедия

7

ПОЧЕМУ ПРОВАЛИЛСЯ ВАВИЛОНСКИЙ ПРОЕКТ?

На всей земле был один язык и одно наречие. Двинувшись с востока, они нашли в земле Сennaар равнину и поселились там. И сказали друг другу: наделаем кирпичей и обожжем огнем. И стали у них кирпичи вместо камней, а земляная смола вместо извести. И сказали они: построим себе город и башню, высотою до небес, и сделаем себе имя прежде, нежели расеемся по лицу всей земли. И сошел Господь посмотреть город и башню, которые строили сыны человеческие. И сказал Господь: вот, один народ, и один у всех язык; и вот что они начали делать, и не отстанут они от того, что задумали делать; сойдем же и смешаем там язык их, так чтобы один не понимал речи другого. И рассеял их Господь оттуда по всей земле; и они перестали строить город (и башню).

Книга Бытия 11:1-8

УПРАВЛЕНЧЕСКИЙ АУДИТ ВАВИЛОНСКОГО ПРОЕКТА

Согласно преданию в книге Бытия, Вавилонская башня была вторым крупным инженерным предприятием человека после Ноева ковчега. И она стала первым инженерным фиаско.

Данная история глубока и поучительна на нескольких уровнях. Давайте, однако, проанализируем ее с точки зрения инженерного проекта и посмотрим, какие управленческие уроки можно извлечь. Как много у Вавилонского проекта было предпосылок для успеха? Имелись ли у строителей:

- *Четкая миссия?* Да, хотя и наивно невозможная. Проект провалился задолго до того, как столкнулся с этим фундаментальным ограничением.
- *Рабочая сила?* В достатке.
- *Материалы?* В Месопотамии в изобилии встречаются глина и битум.
- *Достаточно времени?* Да, на нехватку времени даже никаких намеков.
- *Соответствующая технология?* Да, пирамидальная или коническая структура, в сущности, является стабильной и хорошо распределяет сжимающую нагрузку. Очевидно, искусство каменной кладки было хорошо изучено. Проект провалился до того, как столкнулся с технологическими ограничениями.

Но в чем причина неуспеха, ведь строители ни в чем не нуждались? Или все же чего-то не хватало? Двух вещей — *коммуникации* и, как ее следствия, *организации*. Они не могли разговаривать друг с другом и, следовательно, координировать свои действия. Когда координация не удалась, работа остановилась. Между строк под-
разумевается, что недостаток коммуникации привел к спорам,

плохим чувствам и групповой конкуренции. Вскоре кланы начали расходиться, предпочитая изоляцию спорам.

КОММУНИКАЦИЯ В БОЛЬШОМ ПРОЕКТЕ

Ситуация повторяется и сегодня. Катастрофа с графиком, функциональная несогласованность и системные ошибки возникают потому, что левая рука не знает, что делает правая. По ходу работы несколько команд медленно меняют функции, размеры и скорость своих собственных программ, а также явно или неявно меняют свои допущения об имеющихся входах и использовании выходов.

Например, имплементатор оверлейной функции может столкнуться с проблемами и снизить ее скорость, полагаясь на статистику, которая показывает, как редко эта функция будет возникать в прикладных программах. А тем временем его сосед, возможно, занимается дизайном главной части супервизора, которая критически зависит от скорости этой функции. Это изменение скорости само по себе становится серьезным изменением спецификации, и о нем нужно провозгласить широко и взвесить его с точки зрения системы.

Как же тогда команды должны обмениваться информацией друг с другом? Всеми возможными способами:

- *Неофициально.* Хорошая телефонная служба и четкое определение межгрупповых зависимостей обеспечат сотни звонков, от которых зависит общее толкование письменных документов.
- *Совещания.* Неоценимое значение имеют регулярные совещания по проекту, на которых одна команда за другой проводит

технические брифинги. Таким образом удастся избежать сотни мелких недоразумений.

- *Рабочая книга.* В самом начале должна быть запущена формальная рабочая книга проекта. Она заслуживает отдельного раздела.

РАБОЧАЯ КНИГА ПРОЕКТА

ЧТО. Рабочая книга проекта — это не столько отдельный документ, сколько структура, налагаемая на документы, которые проект так или иначе будет производить.

Все документы проекта должны быть частью этой структуры. Она включает в себя целевые критерии, внешние спецификации, интерфейсные спецификации, технические стандарты, внутренние спецификации и административные меморандумы.

ПОЧЕМУ. Технологический документ практически вечен. Если изучить генеалогию клиентского справочного руководства в части аппаратного или программного обеспечения, то можно проследить не только идеи, но и многие из тех самых предложений и абзацев вплоть до первых меморандумов, которые предлагают продукт или объясняют первый дизайн. Для составителя технической документации пузырек с клеем такой же действенный инструмент, как и перо.

Поскольку это так и поскольку завтрашние документации по качеству продукции вырастут из сегодняшних заметок, очень важно правильно выстроить структуру документации. Ранний дизайн рабочей книги проекта обеспечивает мастерское структурирование документации. Более того, установление структуры позволяет составленные позднее документы оформить в виде отрывков, которые вписываются в эту структуру.

Вторая причина существования рабочей книги проекта состоит в контроле распределения информации. Это подразумевает не ограничение информации, а, наоборот, возможность доводить ее до всех сотрудников, которые в ней нуждаются.

Для начала следует пронумеровать все меморандумы так, чтобы были доступны упорядоченные списки заголовков и каждый работник мог видеть, есть ли у него то, что он хочет. Этим организация рабочей книги далеко не ограничивается, устанавливая древовидную структуру меморандумов. Древовидная структура позволяет поддерживать распределительные списки по поддеревьям, если это желательно.

МЕХАНИКА. Как и во многих задачах управления программными проектами, проблема технических меморандумов усложняется нелинейным образом по мере увеличения объема данных. С 10 людьми документы могут быть просто пронумерованы. Для 100 человек часто достаточно нескольких линейных последовательностей. С тысячей людей, неизбежно разбросанных по нескольким площадкам, *потребность* в структурированной книге возрастает, а *размер* книги увеличивается. Как же тогда должна работать такая механика?

Думаю, что это было хорошо сделано в проекте OS/360. Потребность в грамотно структурированной рабочей книге была настоятельно рекомендована О. С. Локеном (O. S. Locken), который увидел ее эффективность в своем предыдущем проекте, операционной системе 1410-7010.

Мы быстро решили, что *каждый* программист должен видеть *весь* материал, то есть иметь копию рабочей книги в своем кабинете.

Решающее значение имеет ее своевременное обновление. Книга должна быть актуальной. Это очень трудно сделать, если для внесения изменений должны быть перепечатаны целые документы.

В переплетенной книге следовало заменить только страницы. У нас была компьютерная система редактирования текста, и для своевременного сопровождения она оказалась неоценимой. Офсетные формы готовились непосредственно на компьютерном принтере, а оборотное время* составляло менее суток. Однако у получателя всех этих обновленных страниц выявилась проблема с усвоением материала. Когда он впервые получает измененную страницу, он хочет знать: что изменилось? Когда он обращается к ней позже, он хочет знать: каким на сегодня является определение?

Последняя потребность удовлетворяется непрерывно сопровождаемым документом. Выделение изменений требует других шагов. Во-первых, на странице необходимо отметить измененный текст, например вертикальной полосой на полях рядом с каждой измененной строкой. Во-вторых, вместе с новыми страницами необходимо распространить краткое, отдельно написанное резюме изменений, в котором изменения и замечания перечислены по мере их значимости.

Мы не работали и 6 месяцев, как столкнулись с еще одной проблемой в нашем проекте. Рабочая книга была около пяти футов толщиной! Если бы мы сложили 100 копий, которые обслуживали программистов в наших офисах в манхэттенском здании Time-Life, то они превысили бы высоту самого здания. Более того, сводка ежедневных изменений имела в толщину в среднем два дюйма, что составляло около 150 страниц, которые должны были быть совмещены с целым. Ведение рабочей книги стало занимать значительное время каждого рабочего дня.

* Оборотное время (*turn around time*) — это период времени, необходимый для выполнения определенного процесса или задачи с момента, когда оно было официально затребовано. — *Примеч. пер.*

В этот момент мы переключились на микрофиши*, и это изменение сэкономило миллион долларов, даже с учетом стоимости считывателя микрофишей для каждого офиса. Мы смогли организовать отличный оборот на микрофишах; рабочая тетрадь сократилась с трех кубических футов до одной шестой кубического фута, и что самое важное, обновления появились кусками по 100 страниц, что в 100 раз уменьшило проблему совмещения.

Микрофиши имеют свои недостатки. С точки зрения менеджера, ручное совмещение бумажных страниц обеспечивало *прочтение* изменений, что и было целью рабочей книги. Микрофиши облегчили обслуживание рабочей книги, при условии, что обновление не распространялось вместе с бумажным документом, в котором эти изменения перечислялись.

Кроме того, читатель не имеет возможности легко выделять, отмечать и комментировать микрофиши. Документы, с которыми читатель взаимодействовал до этого, являются более эффективными для автора и более полезными для читателя.

В целом, думаю, что микрофиши сами по себе удачный механизм, и я бы рекомендовал его для использования в очень крупных проектах вместо бумажной книги.

КАК БЫ ЭТО БЫЛО СДЕЛАНО СЕГОДНЯ? С учетом сегодняшних технологий думаю, что предпочтительный метод — держать рабочую книгу в файле прямого доступа, помеченном полосами изменений и датами ревизий. Каждый пользователь будет обращаться к нему с дисплейного терминала (печать является слишком медленной). Сводка изменений, готовящаяся ежедневно, будет храниться в виде стека LIFO в фиксированной точке доступа. Программист, вероят-

* Микрофиша — документ в виде микроформы на прозрачной форматной пленке с последовательным расположением кадров в несколько рядов. Использовались в библиотеках, архивах и проектных бюро для сокращения физических объемов хранилищ документальной информации. — *Примеч. ред.*

но, будет читать ее ежедневно, но если пропустит день, ему лишь придется прочитать материал следующего дня. Читая сводку изменений, он может прерываться для сверки с самим измененным текстом.

Обратите внимание, что сама книга не изменилась. Это по-прежнему совокупность всей проектной документации, структурированной в соответствии с тщательным дизайном. Единственное изменение состоит в механике распределения и консультаций. Дуглас Энгельбарт (D. C. Engelbart) и его коллеги из Стэнфордского исследовательского института создали такую систему и используют ее для сборки и ведения документации для сети ARPANET.

Дэвид Парнас (D. L. Parnas) из Университета Карнеги-Меллона предложил еще более радикальное решение. Его тезис заключается в том, что программист наиболее эффективен, если он огражден от подробностей конструкции частей системы, отличных от его собственных. Такой подход предполагает, что все интерфейсы полностью и точно определены. Хотя такой дизайн, безусловно, является здравым, зависимость от его идеального исполнения ведет к катастрофе. Хорошая информационная система не только выявляет ошибки интерфейса, но и стимулирует их исправление.

ОРГАНИЗАЦИЯ В КРУПНОМ ПРОЕКТЕ

Если в проекте участвует n работников, то существует $(n^2 - n)/2$ интерфейсов, через которые может осуществляться коммуникация, и потенциально почти 2^n команд сотрудников, внутри которых должна происходить координация. Цель организации состоит в том, чтобы уменьшить объем необходимой коммуникации и координации; следовательно, организация представляет собой радикальную атаку на рассмотренные выше проблемы коммуникации.

Вполне обойтись без коммуникаций позволяют *разделение труда* и *специализация функций*. Древовидная структура организаций отражает меньшую потребность в детальной коммуникации при применении разделения и специализации труда.

Фактически древовидная организация возникает как структура полномочий и ответственности. Принцип, согласно которому ни один человек не может служить двум господам, диктует, чтобы структура полномочий была древовидной. Но коммуникационная структура не так ограничена, и дерево лишь отчасти упрощает коммуникационную структуру, которая представляет собой сеть. Неадекватность древовидного упрощения приводит к возникновению штатных групп, оперативных сил, комитетов и даже матричных организаций, используемых во многих инженерных лабораториях.

Давайте рассмотрим древовидную организацию программирования и проанализируем существенные свойства, которыми должно обладать любое поддерево в целях эффективности. Таковыми являются:

- Миссия.
- Продюсер.
- Технический директор или архитектор.
- График работ.
- Разделение труда.
- Определения интерфейсов между частями.

Все это является очевидным и традиционным, за исключением различия между продюсером и техническим директором. Давайте сначала рассмотрим эти две роли, а затем их взаимосвязи.

Какова роль продюсера? Он собирает команду, распределяет работу и устанавливает график. Он приобретает и продолжает приобретать

необходимые ресурсы. Это означает, что главенствующая часть его роли состоит в коммуникации вне команды, вверх и в сторону. Он устанавливает схему коммуникации и отчетности внутри команды. Наконец, он обеспечивает выполнение графика, перемещая ресурсы и меняя организацию в соответствии с меняющимися обстоятельствами.

Что насчет технического директора? Он задумывает дизайн, который будет построен, выявляет его подчасти, определяет, как дизайн будет выглядеть со стороны, и очерчивает схему его внутренней структуры. Он обеспечивает единство и концептуальную целостность проекта и таким образом способствует ограничению сложности системы. По мере возникновения отдельных технических проблем он изобретает для них решения либо по мере необходимости меняет дизайн системы. Он, по выражению Эла Каппа (Al Carr), является «своим человеком в дурно пахнущих делах». Его коммуникации сосредоточены главным образом внутри команды. Его работа почти полностью является технической.

Теперь ясно, что таланты, необходимые для этих двух ролей, являются совершенно разными. Способности встречаются в разных сочетаниях, и отношения между продюсером и директором должны определяться теми конкретными сочетаниями, которыми они обладают. Организации должны создаваться вокруг имеющихся людей, а не вписывать людей в чисто теоретическую организационную структуру.

Возможны три взаимосвязи, и все три встречаются в успешной практике.

ПРОДЮСЕР И ТЕХНИЧЕСКИЙ ДИРЕКТОР МОГУТ БЫТЬ ОДИМ И ТЕМ ЖЕ ЧЕЛОВЕКОМ.

Это легко работает в очень малых командах, возможно, от трех до шести программистов. На крупных проектах это очень редко

работает по двум причинам. Во-первых, человек с сильным менеджерским талантом и сильным техническим талантом редко встречается. Мыслители встречаются редко; практики еще реже, а мыслители-практики являются самыми редкими.

Во-вторых, в крупном проекте каждая из ролей обязательно требует полного рабочего дня или более. Продюсер едва сможет делегировать свои обязанности в достаточном объеме, чтобы выкроить время на техническую работу. Директор не может делегировать свои полномочия без ущерба для концептуальной целостности дизайна.

ПРОДЮСЕР МОЖЕТ БЫТЬ БОССОМ, А ДИРЕКТОР — ЕГО ПРАВОЙ РУКОЙ.

Здесь трудность заключается в том, чтобы установить полномочия директора принимать технические решения, не влияя на его время, как если бы он был включен в менеджерскую цепочку командования.

Очевидно, что продюсер должен провозгласить технические полномочия директора и должен поддерживать их в чрезвычайно высокой доле возникающих случаев. Для этого необходимо, чтобы продюсер и директор одинаково смотрели на фундаментальную техническую философию; важно обсуждать главные технические вопросы в частном порядке, прежде чем они действительно станут насущными; и продюсер должен иметь уважение к техническому мастерству директора.

Что менее очевидно, продюсер может расставлять едва уловимые акценты с символами статуса (размер офиса, ковер, мебель, принадлежности и т. д.), провозглашая, что директор, хотя и находится вне управленческой линии, является источником полномочий при принятии решений.

Так при правильной организации можно получить хороший эффект. К сожалению, это редко срабатывает на практике. У менед-

жеров проектов плохо получается использовать технический гений людей, которые не сильны в управлении.

ДИРЕКТОР МОЖЕТ БЫТЬ БОССОМ, А ПРОДЮСЕР — ЕГО ПРАВОЙ РУКОЙ.

Роберт Хайнлайн в книге «Человек, который продал Луну» красочно описывает такую расстановку:

Костер уткнулся лицом в ладони, затем снова поднял глаза:

— Я понимаю. И знаю, что следует сделать. Однако стоит мне заняться техническими вопросами, какой-нибудь чертов кретин требует от меня распоряжений — о грузовиках, телефонах, черте, дьяволе!.. Ох, извините, мистер Харриман. Думаю, все же справлюсь со всем этим.

— Ну-ну, не расстраивайся, — ласково сказал Харриман. — Ты последнее время недосыпал, верно? Так вот, давай-ка разыграем нашего Фергюссона. На несколько дней я сам сяду за твой стол и устрою все так, чтобы тебя не отрывали от работы подобными вещами. Я хочу, чтобы твоя голова была занята векторами реакции, эффективностью топлива и напряжением конструкций, а не контрактами об аренде грузовиков.

Выглянув в приемную, Харриман увидел там странного типа: не то клерка, не то уборщика.

— Эй, ты! Поди-ка сюда!..

Тип здорово удивился, встал и вошел в кабинет.

— Слушаю вас.

— Вот этот стол, что в углу, и все хозяйство, которое на нем, немедленно перенести в свободный кабинет на этом же этаже.

Тот поднял брови:

— А кто вы, собственно, такой, хотел бы я знать?

— Какого...

— Выполняйте, Уэбер, — приказал Костер.

— И чтоб через двадцать минут было сделано! — добавил Харриман. — Шевелись.

Перейдя к другому столу, он взялся за телефон и позвонил в управление «Скайуэйз».

— Джим, там у тебя Джек Беркли твой далеко? Сделай ему отпуск и пришли ко мне, в «Петерсон Филд», спецрейсом, немедленно. Я хочу, чтобы корабль с ним на борту стартовал через десять минут после того, как ты повесишь трубку. Вещи пошлете вслед за ним.

Некоторое время Харриман слушал, затем сказал:

— Да не развалится без него твоя контора! А если развалится, значит, ты даром деньги получаешь. Ладно, ладно, когда поймаешь меня в следующий раз, можешь разок пнуть под зад, однако Джока высылай. Пока.

Он проверил, как выполнен приказ о переносе рабочего стола Костера в другой кабинет, проследил, чтобы там не было телефона, и задним числом сообразил приказать перетащить к Костеру и диван.

— Проектор, чертежную машину, стеллажи для книг и прочее установим вечером. Ты только составь список, что тебе нужно для работы.

Вернувшись в кабинет номинального главы проекта, он счастливо улыбнулся, засучил рукава и взялся за дело, стараясь понять, в каком состоянии эта лавочка и что в ней не так.

Часа через четыре он повел прибывшего Джока Беркли к Костеру. Главный инженер спал за столом, уронив голову на руки. Харриман на цыпочках двинулся к дверям, но Костер вскинулся:

— Фу-ты! Извините, мистер Харриман. Отключился...

— Для этого у тебя есть диван, — заметил Харриман, — на нем удобней. Боб, познакомься, это — Джок Беркли, твой покорный слуга. Ты остаешься главным инженером и высшим, неоспоримым начальством, а Джок — властелином всего прочего. С этого момента можешь ни о чем не беспокоиться, исключая один пустяк — постройку лунного корабля.

— Только об одном вас попрошу, мистер Костер, — сказал Беркли, пожимая ему руку, — можете делать через мою голову все, что хотите... и все, что надо для вашей техники, только, ради Всевышнего, все записывайте, чтобы я был в курсе происходящего. Я установлю на вашем столе кнопку, которая будет включать магнитофон у меня в столе.

— Замечательно!

Харриман отметил, что Костер как будто помолодел.

— Понадобится что-нибудь, не относящееся к технике, не делайте ничего сами. Только нажмите кнопку да свистните — и все будет сделано. — Беркли покосился на Харримана. — Босс сказал, что хочет поговорить с вами о деле. А я пойду заниматься своим.

С этими словами он вышел.

Харриман сел. Костер последовал его примеру:

— Уф-ф-ф!

— Теперь лучше?

— Здорово выглядит этот ваш Беркли!

— Рад, что тебе он нравится. С этой минуты вы с ним — близнецы-братья. Можешь не беспокоиться, я сам с ним работал. Считаю, что лежишь в приличной больнице. Ты, кстати, где остановился?*

* Роберт Хайнлайн. Человек, который продал Луну / *The Man Who Sold the Moon* (1993, повесть), пер. Д. А. Старков. — *Примеч. ред.*

Этот фрагмент книги вряд ли нуждается в каком-либо аналитическом комментарии. Такую расстановку тоже можно заставить эффективно работать.

Подозреваю, что последняя расстановка лучше всего подходит для небольших команд, как обсуждалось в главе 3 «Хирургическая бригада». Думаю, что продюсер в качестве босса является более подходящей схемой для больших ответвлений действительно крупного проекта.

Вавилонская башня была, пожалуй, первым инженерным фиаско, но не последним. Решающее значение для успеха имеют коммуникация и ее последующая организация. Методы коммуникации и организации требуют от менеджера тщательных размышлений и такой же кадровой компетентности, как и сама технология программного обеспечения.



Дуглас Кроквелл. «Рут называет свою попытку».
Мировая серия, 1932 год. Приводится с разрешения
журнала Esquire и Дугласа Кроквелла, © 1945
(возобновлено в 1973 году) компанией Esquire, Inc.,
и любезно предоставлено Национальным музеем бейсбола.

8

ПОПЫТКИ ИЗМЕРИТЬ

Практика — лучший учитель.

Публий

Опыт — дорогой учитель,
но для глупцов иного нет.

Альманах Бедного Ричарда

Сколько времени займет работа по программированию системы? Сколько потребуются усилий? Как это можно оценить?

Ранее я предлагал соотношения, которые, по-видимому, применимы к планированию времени, кодирования, тестирования компонентов и тестирования системы. Во-первых, следует сказать, что *не* следует оценивать всю задачу, оценивая только часть, относящуюся к написанию программ, а затем применяя соотношения. Написание кода составляет примерно одну шестую задачи, и ошибки в его оценке или в соотношениях могут привести к нелепым результатам.

Во-вторых, необходимо сказать, что данные, относимые к разработке изолированных малых программ, неприменимы к продуктам систем программирования. К примеру, для программы, насчитывающей 3200 слов, Сакман, Эриксон и Грант оценивают суммарное время написания программ и отладки для одного программиста в 178 часов, что экстраполируется до 35800 операторов в год. Сама программа вдвое меньшего размера занимала менее одной четверти времени, а экстраполированная продуктивность составляла почти 80 тысяч инструкций в год.¹ Необходимо добавить время на планирование, документирование, тестирование, системную интеграцию и обучение. Линеинно экстраполировать показатели забега в спринте бессмысленно. Экстраполяция времени забега на 100 ярдов показывает, что человек может пробежать милю менее чем за три минуты. Прежде чем отказаться от этих данных, отметим, что и для не совсем сравнимых задач они показывают, что объем работы растет как степенная функция размера, *даже* без учета процесса отмена информацией (кроме программиста с собственной памятью).

Рисунок 8.1 иллюстрирует печальную историю. На нем отображены результаты, полученные в исследовании, проведенном Нанусом (Nanus) и Фарром (Farr)² в корпорации System Development Corporation (SDC). Оно показывает экспоненту 1,5; то есть

$$\text{усилие} = (\text{константа}) \times (\text{число операторов})^{1,5}$$

Еще одно исследование SDC, о котором сообщает Вайнвурм (Weinwurm),³ также показывает экспоненту, равную примерно 1,5.

Было проведено несколько исследований продуктивности программистов и предложено несколько методов оценивания. Морин (Morin) подготовил обзор опубликованных данных.⁴ Здесь я приведу лишь несколько выдержек, которые выглядят особенно показательными.

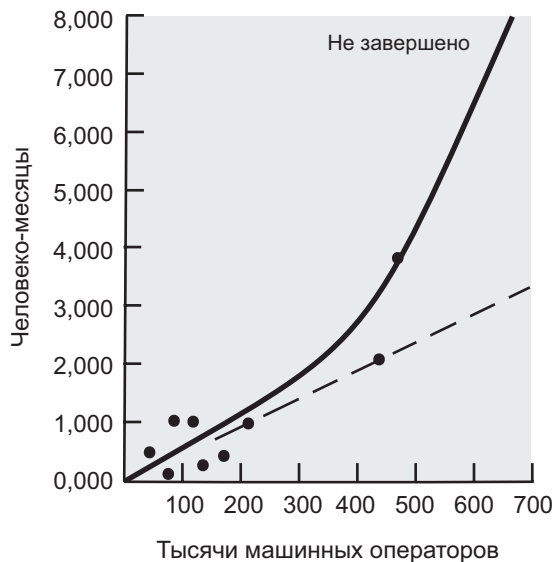


Рис. 8.1. Затраты на программирование как функция размера программы

ДАННЫЕ ПОРТМАНА

Чарльз Портман (Charles Portman), менеджер отдела программного обеспечения ICL, *Computer Equipment Organization* (Северо-Запад) в Манчестере, предлагает свой взгляд на проблему.⁵

Он обнаружил, что его команды соблюдают сроки примерно наполовину — каждая задача занимала примерно вдвое больше времени, чем предполагалось. Оценки были очень тщательными, выполненными опытными командами, оценивающими человеко-часы для нескольких сотен задач на диаграмме PERT. Когда проявился шаблон отставания от графика, он попросил их вести тщательный ежедневный журнал учета использования времени. Выявилось, что ошибка оценивания могла полностью объясняться тем, что его команды реализовывали только 50% рабочей недели на программирование и отладку. Время простоя машин, более приоритетные короткие несвязанные работы, совещания, работа по созданию документации, корпоративные дела, болезни, личное время и т. д. объясняло все остальное. Одним словом, в оценках делалось нереалистичное допущение о числе технических часов работы в расчете на человеко-год. Мой собственный опыт вполне подтверждает его вывод.⁶

ДАННЫЕ ЭЙРОНА

Джоэл Эйрон (Joel Aron), менеджер по системным технологиям в IBM в Гейтерсберге, штат Мэриленд, изучил продуктивность программистов во время работы с девятью крупными системами (в двух словах, под *крупными* подразумевается более 25 программистов и 30 000 поставляемых операторов).⁷ Он разделяет такие системы в соответствии со взаимодействиями между программами (и частями системы) и выводит продуктивность следующим образом:

Очень мало взаимодействий	10 000 операторов на человеко-год
Немного взаимодействий	5000
Много взаимодействий	1500

Человеко-годы не включают в себя деятельность по поддержке и тестированию системы, только дизайн и программирование. Когда эти показатели разбавляются в два раза, с учетом тестирования системы, они близко соответствуют данным Харра.

ДАННЫЕ ХАРРА

Джон Харр (John Harr), менеджер по программированию для электронной коммутационной системы компании *Bell Telephone Laboratories*, сообщил о своем опыте и опыте своих коллег в докладе на компьютерной конференции *Spring Joint Computer Conference* 1969 года.⁸ Эти данные показаны на рис. 8.2, 8.3 и 8.4.

Из них рис. 8.2 является наиболее подробным и наиболее полезным. Первые два задания — это в основном программы управления; вторые два — по сути, трансляторы языка. Продуктивность выражается в отлаженных словах в расчете на один человеко-год. Сюда входит программирование, тестирование компонентов и тестирование системы. Но неясно, какая часть усилий по планированию, сопровождению машин, написанию и т. п. сюда внесена.

	Число программных блоков	Число программистов	Затрачено лет	Человеко-лет	Количество слов в программе	Слов / человеко-год
Операционная	50	83	4	101	52 000	515
Обслуживающая	36	60	4	81	51 000	630
Компилятор	13	9	2¼	17	38 000	2230
Транслятор (ассемблер)	15	13	2½	11	25 000	2270

Рис. 8.2. Сводка по четырем программным заданиям ESS № 1

Продуктивность схожим образом делятся на две классификации: классификация для управляющих программ составляет около 600 слов на человеко-год; классификация для трансляторов составляет около 2200 слов на человеко-год. Обратите внимание, что все четыре программы имеют сходный размер — разница лишь в размере команд, продолжительности времени и числе модулей. Что является причиной, а что следствием? Управляющие программы требовали больше людей, потому что были сложнее? Или им требовалось больше модулей и больше человеко-месяцев, потому что на этот участок назначалось больше людей? Занимали ли они больше времени из-за большей сложности или же потому, что было назначено больше людей? Никто не ответит с уверенностью. Управляющие программы, разумеется, были сложнее. Если оставить в стороне эти неопределенности, то цифры описывают реальную производительность при создании больших систем, и поэтому представляют ценность.

На рис. 8.3 и 8.4 показаны некоторые интересные данные о темпах программирования и отладки по сравнению с предсказываемыми.

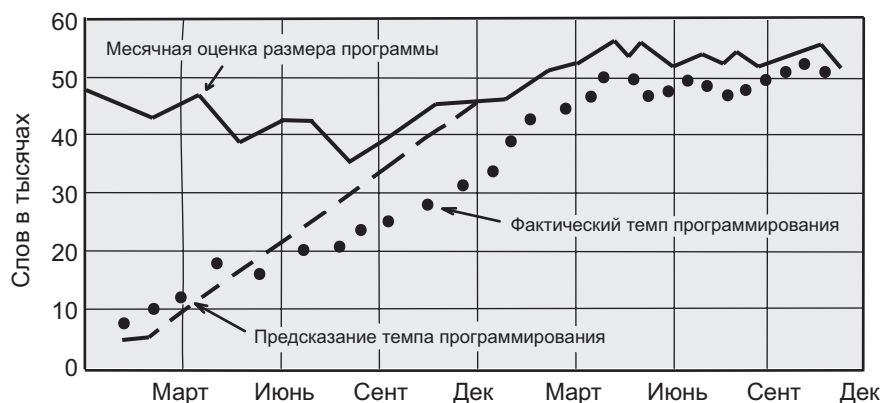


Рис. 8.3. Предсказываемые и фактические темпы программирования ESS

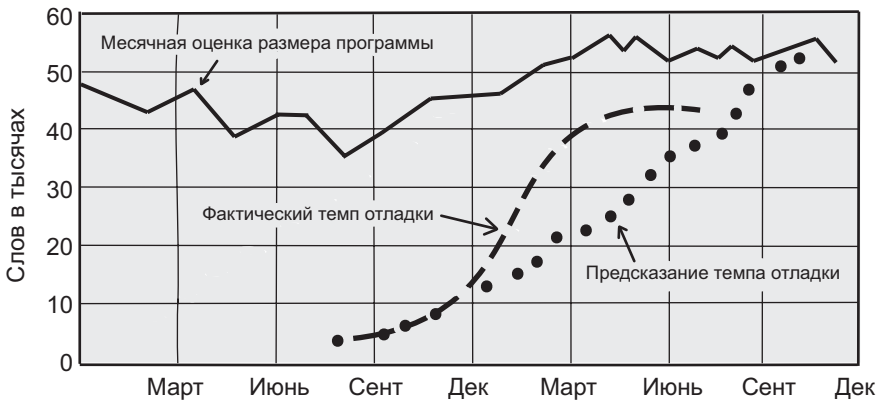


Рис. 8.4. Предсказываемые и фактические темпы отладки ESS

ДАННЫЕ OS/360

Опыт IBM OS/360, хотя и не представленный так же подробно, как данные Харра, это подтверждает. Группы по работе с управляющими программами показывали продуктивность в диапазоне 600–800 отлаженных инструкций в расчете на человеко-год. Группы по работе с трансляторами языков достигали продуктивности в размере 2000–3000 отлаженных инструкций в расчете на человеко-год. Сюда входит выполняемое группой планирование, а также кодирование теста компонентов, тестирование системы и некоторые вспомогательные действия. Насколько я могу судить, они сопоставимы с данными Харра.

Данные Эйрона, данные Харра и данные OS/360 — все они подтверждают поразительные различия в продуктивности, связанные со сложностью и трудностью самой работы. Мой вывод среди всей этой сложности в оценках состоит в том, что компиляторы в 3 раза хуже обычных пакетных прикладных программ, а операционные системы в 3 раза хуже компиляторов.⁹

ДАННЫЕ КОРБАТО

И данные Харра, и данные OS/360 относятся к программированию на языке ассемблера. По-видимому, было опубликовано мало данных о продуктивности системного программирования с использованием языков более высокого уровня. Однако Корбато (Corbato) из проекта MAC Массачусетского технологического университета сообщает о средней продуктивности 1200 строк отлаженных операторов языка PL/1 в расчете на человеко-год в системе MULTICS (от 1 до 2 миллионов слов).¹⁰

Это число является очень заманчивым. Как и другие проекты, MULTICS включает в себя управляющие программы и трансляторы языков. Как и другие, он производит продукт системного программирования, протестированный и задокументированный. Данные, по-видимому, сопоставимы с точки зрения вида прилагаемых усилий. А показатель продуктивности является хорошим средним арифметическим между продуктивностью по работе с управляющей программой и продуктивностью по работе с трансляторами других проектов.

Но число Корбато — это *строки* в расчете на человеко-год, а не *слова*! Каждая инструкция в его системе соответствует примерно 3–5 словам рукописного кода! Это позволяет сделать два важных вывода.

- Продуктивность, измеренная в элементарных операциях, оказывается постоянной, что кажется разумным, если учитывать, сколько времени нужно думать над оператором и сколько ошибок может в нем быть.
- Продуктивность программирования может быть увеличена в 5 раз при использовании подходящего языка высокого уровня.¹¹



9

НЕПОСИЛЬНЫЙ ГРУЗ

Автору стоит присмотреться к Ною и... поучиться на примере Ковчега, как в очень маленькое пространство втиснуть очень много.

Сидни Смит. Эдинбургский обзор

ПРОСТРАНСТВО ПРОГРАММЫ КАК СТОИМОСТЬ

Насколько она велика? Помимо времени выполнения, пространство, занимаемое программой, влияет на стоимость в первую очередь. Это справедливо даже для проприетарных программ, где пользователь платит автору гонорар, составляющий, по сути, часть бюджета на разработку. Возьмем интерактивную программно-информационную систему IBM APL. Она арендуется за \$400 в месяц, и при ее использовании она занимает не менее 160 Кбайт памяти. В системе Model 165 память арендуется примерно за \$12 за килобайт в месяц. Если программа доступна полный рабочий день, то пользование программой составляет \$400 за аренду программного обеспечения и \$1920 за аренду памяти. Если система APL используется только четыре часа в день, то стоимость составит \$400 за аренду программного обеспечения и \$320 за аренду памяти в месяц.

Часто можно услышать выражения ужаса, что 2 Мб машина выделяет 400 Кбайт под ее операционную систему. Это так же глупо, как критиковать Boeing 747 за того, что он стоит \$27 миллионов. Нужно также спросить: «А что он делает?» Что можно получить в виде простоты использования и производительности (за счет эффективного использования системы) за потраченные таким образом доллары? Могли ли \$4800 в месяц, вложенные таким образом в аренду памяти, быть потрачены плодотворнее на другое оборудование, на программистов, на прикладные программы?

Системный дизайнер распределяет часть суммарных аппаратных ресурсов для хранения в памяти резидентных программ, когда он считает, что это полезнее для пользователя именно в такой форме, чем сумматоры, диски и т. д. Поступить иначе было бы крайне безответственно. И о результате надо судить в целом. Никто не может критиковать систему программирования за размер как таковой и в то же время последовательно выступать за более тесную интеграцию аппаратного и программного обеспечения.

Поскольку размер является для пользователя такой крупной частью стоимости продукта системы программирования, разработчик должен устанавливать целевые показатели размера, контролировать размер и разрабатывать методику сокращения размера, так же как разработчик аппаратного обеспечения устанавливает целевые показатели количества компонентов, контролирует количество компонентов и разрабатывает методику сокращения количества. Как и любая стоимость, размер сам по себе не является плохим, но нежелательный размер таковым является.

КОНТРОЛЬ РАЗМЕРОВ

Для менеджера проекта контроль размеров — это отчасти техническая работа, отчасти управленческая. Установление размеров предлагаемых систем требует предварительного изучения пользователей и их потребностей. Затем эти системы должны быть подразделены и каждому компоненту задан размерный показатель. Поскольку компромиссы между размером и скоростью проявляются довольно большими скачками, установление целевых показателей размера является сложным делом, требующим знания о доступных компромиссах в каждой части. Мудрый менеджер также припасет резерв, чтобы обращаться к нему в ходе работы.

В OS/360, несмотря на то что все это было сделано очень тщательно, пришлось с болью усвоить и другие уроки.

Во-первых, недостаточно установить целевые показатели размера для ядра; необходимо бюджетировать все аспекты размера. Большинство предыдущих операционных систем располагалось на ленте, и долгое время поиска ленты означало, что не было соблазна использовать ее небрежно для обращения к программе. OS/360 была дисково-резидентной, как и ее непосредственные предшественники, операционная система Stretch и дисковая операционная

система 1410-7010. Ее создатели возрадовались свободе дешевого доступа к дискам. И в результате первоначальный результат оказался катастрофическим для производительности.

Устанавливая размеры ядра для каждого компонента, мы при этом не устанавливали бюджеты доступа. Как и следовало ожидать, программист, выходявший за рамки выделенной ему памяти, разбивал программу на оверлей*. Этот процесс сам по себе увеличивает суммарный размер программы и замедляет ее исполнение. Самое главное, что наша система контроля этого не измеряла и не улавливала. Каждый сообщал, сколько *ядер* он использует, и поскольку этот показатель был в пределах целевого, никто не беспокоился.

К счастью, еще в начале процесса разработки мы начали использовать симулятор производительности OS/360. Первый результат свидетельствовал о большой беде. Fortran H, на системе Model 65 с барабанами, симулировал компиляцию со скоростью пять инструкций в минуту! Разбор проблемы показал, что каждый модуль управляющих программ делал очень много обращений к дискам. Даже высокочастотные модули супервизора совершали множество обращений к диску, и результат был вполне аналогичен просеиванию страниц.

Первый вывод очевиден: установить *общие* размерные бюджеты, а также бюджеты резидентного пространства; установить бюджеты на доступ к дисковому хранилищу, а также к его размеру.

Следующий урок был очень похож. Пространственные бюджеты определялись до того, как функциональность была точно определена для каждого модуля. В результате любой программист с размерной проблемой проверял свой код на наличие того, что он мог

* Overlay (оверлей) — метод программирования, позволяющий создавать программы, занимающие больше памяти, чем установлено в системе, за счет динамической загрузки с диска частей программного кода и данных. — *Примеч. ред.*

бы выбросить. Поэтому буферы под управлением управляющей программы стали частью пользовательского пространства памяти. Что еще хуже, так же поступали все виды блоков управления, в результате чего безопасность и защита системы оказались под угрозой.

Поэтому второй вывод тоже совершенно ясен: при задании размера модуля нужно точно определить, что он должен делать.

Третий и более глубокий урок проявляется через этот опыт. Проект был достаточно крупным, а коммуникация с менеджером достаточно слабой, что побуждало многих членов команды чувствовать себя конкурсантами, зарабатывающими зачетные очки, а не разработчиками, создающими программные продукты. Каждый субоптимизировал свою работу для достижения своих целей; немногие останавливались для того, чтобы подумать о суммарном эффекте для клиента. Это нарушение ориентации и коммуникации является серьезной опасностью для крупных проектов. На протяжении всего процесса имплементации системные архитекторы должны постоянно следить за сохранением целостности системы. Однако за пределами этого механизма поддержания порядка стоит вопрос о позиции самих имплементаторов. Воспитание общесистемного, ориентированного на пользователя отношения вполне может быть самой важной функцией менеджера по программированию.

ПРОСТРАНСТВЕННЫЕ МЕТОДИКИ

Никакое количество пространственного бюджетирования и контроля не может сделать программу маленькой. Это требует изобретательности и мастерства.

Очевидно, что бóльшая функциональность означает больше пространства при константной скорости. И поэтому первый аспект мастерства заключается нахождении компромисса между функ-

циональностью и размером. Здесь возникает ранний и глубокий политический вопрос. В какой мере поиск компромисса может быть оставлен пользователю? Можно разработать программу с большим числом необязательных функциональных свойств, каждое из которых занимает не много места. Можно спланировать генератор, который будет принимать список опций, и подстраивать программу под него. Но для любого конкретного набора опций более монолитная программа будет занимать меньше места. Это, скорее, похоже на автомобиль: если подсветка, прикуриватель и часы оцениваются вместе как одна опция, то пакет будет стоить меньше, чем если выбирать каждую по отдельности. Поэтому дизайнер должен решить, насколько мелкозернистым будет выбор опций со стороны пользователя.

В процессе дизайна системы для диапазона размеров памяти возникает еще один вопрос. Диапазон приспособляемости нельзя сделать произвольно широким — даже при разбиении программы на очень мелкие модули. В маленькой системе большинство модулей перегружается. Значительная часть резидентной памяти маленькой системы должна быть отведена для временной или страничной памяти, в которую загружаются другие части. Ее размер ограничивает размер каждого модуля. А разбиение функций на малые модули имеет свою стоимость как по производительности, так и по пространству. Поэтому крупная система, которая может позволить себе транзитную область в 20 раз большую, за счет этого экономит только на доступе к ней. И по-прежнему будет страдать как в скорости, так и в пространстве, потому что размер модуля очень мал. Этот недостаток накладывает лимит на максимально эффективную систему, которая может быть сгенерирована из модулей малой системы.

Второй аспект мастерства — это пространственно-временные компромиссы. Для заданной функции чем больше пространства, тем быстрее. Это верно в удивительно большом диапазоне. Именно

этот факт делает возможным установление пространственных бюджетов.

Чтобы помочь своей команде принять удачные решения, менеджер может сделать две вещи. Первая — организовать обучение технике программирования, а не просто полагаться на природный ум и предыдущий опыт. Это особенно важно, если речь идет о новом языке или машине. Особенности умелого использования должны быть быстро выучены и широко распространены, возможно, с особыми призами или поощрениями за новую методику.

Во-вторых, следует признать, что программирование имеет свою технологию, и компоненты *должны быть* заготовлены заранее. Каждому проекту нужна записная книжка, полная хороших подпрограмм или макросов для организации очередей, поиска, хеширования и сортировки. Для каждой такой функции записная книжка должна иметь по крайней мере две программы, быструю и сжатую. Развитие такой технологии является важной работой реализации, которая может решаться параллельно с архитектурой системы.

ПРЕДСТАВЛЕНИЕ ДАННЫХ — ЭТО СУТЬ ПРОГРАММИРОВАНИЯ

За пределами мастерства лежит изобретательство, и именно здесь рождаются бережливые, щадящие, быстрые программы. Они почти всегда являются результатом стратегического прорыва, а не тактического ума. Иногда стратегическим прорывом становится новый алгоритм, такой как быстрое преобразование Фурье, предложенное Кули и Тьюки или замена n^2 множества сравнений на $n \log n$ при сортировке.

Гораздо чаще стратегический прорыв будет исходить от переделывания представления данных или таблиц. Именно тут находится

сердце программы. Покажите мне свои блок-схемы и спрячьте диаграммы моделей данных, и я буду оставаться в заблуждении. Покажите мне свои диаграммы моделей данных, и я, скорее всего, не буду нуждаться в ваших блок-схемах, они будут очевидны.

Легко привести кучу примеров мощи представлений данных. Я вспоминаю молодого человека, взявшегося построить сложный консольный интерпретатор для IBM 650. Он в итоге упаковал его на невероятно маленьком пространстве, построив интерпретатор для интерпретатора, при том понимании, что человеческие взаимодействия были медленными и нечастыми, но пространство было дорогим. Элегантный маленький компилятор языка Fortran от Digitek использует очень плотное, специализированное представление для самого кода компилятора, в результате чего внешнее хранилище не требуется. Затрачиваемое при декодировании этого представления время возвращается десятикратно за счет избегания ввода-вывода. (Упражнения, приведенные в конце главы 6 книги Брукса и Айверсона «Автоматическая обработка данных» (Brooks and Iverson, *Automatic Data Processing*¹), включают в себя набор таких примеров, как и многие упражнения Кнута (Knuth)).²

Программист, находящийся в тупике из-за нехватки пространства, часто может добиться лучшего, выпутавшись из своего кода, отступив назад и переосмыслив свои данные. Представление данных действительно *является* существенным аспектом программирования.



У. Бенго. «Сцена в старой библиотеке Конгресса», 1897. Архив Беттмана

10

ДОКУМЕНТАРНАЯ ГИПОТЕЗА

Гипотеза:

Среди потока бумаг небольшое число документов становится критически важным, вокруг них вращается управление каждым проектом. Эти документы являются ключевыми личными инструментами менеджера.

Технология, правила организации и традиции ремесла требуют выполнить некоторое количество бюрократической работы по проекту. Для нового менеджера, только что выросшего из специалистов, все это кажется помехой, ненужным отвлечением и штормовой волной, грозящей его потопить. И действительно, большинство из них именно таковы.

Однако постепенно он приходит к пониманию того, что определенный небольшой комплект этих документов воплощает и выражает большую часть его управленческой работы. Подготовка каждого из них служит основным поводом для сосредоточения мысли и кристаллизации обсуждений, которые в противном случае блуждали бы бесконечно. Сопровождение документа становится его механизмом наблюдения и предупреждения сбоев. Сам документ служит контрольным списком, контролем состояния дел и базой данных для отчетности.

Для того чтобы увидеть, как это должно работать в проекте по созданию ПО, давайте рассмотрим конкретные документы, полезные в других контекстах, и попробуем их обобщить.

ДОКУМЕНТЫ ДЛЯ ПРОЕКТА РАЗРАБОТКИ КОМПЬЮТЕРА

Предположим, разрабатывается машина. Каковы критически важные документы?

ЦЕЛЕВЫЕ КРИТЕРИИ. В них определяется удовлетворяемая потребность и цели, пожелания, ограничения и приоритеты.

СПЕЦИФИКАЦИИ. Это справочное руководство по компьютеру плюс характеристики производительности. Это один из первых документов, создаваемых при предложении нового продукта, и последний завершаемый документ.

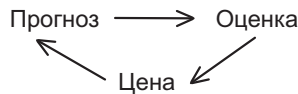
ГРАФИК РАБОТ.

БЮДЖЕТ. Будучи не просто ограничением, бюджет является одним из самых полезных документов менеджера. Существование бюджета вынуждает принимать технические решения, которых в противном случае можно было бы избежать, и, что более важно, он навязывает и разъясняет политические решения.

ОРГАНИЗАЦИОННАЯ СХЕМА.

РАСПРЕДЕЛЕНИЕ ПРОСТРАНСТВА.

ОЦЕНКА, ПРОГНОЗ, ЦЕНЫ. Эти три элемента имеют циклическое замыкание, которое определяет успех или неуспех проекта:



Для создания рыночного прогноза необходимы характеристики производительности и предполагаемые цены. Цифры прогноза вместе с заданным проектом числом компонентов определяют оценку стоимости производства и долю расходов на разработку и фиксированных затрат, приходящихся на одно устройство. Эти расходы, в свою очередь, определяют цены.

Если цены *ниже* тех, которые предполагаются, начинается радостная закрутка спирали успеха. Прогнозы растут, удельные затраты падают, а цены падают еще больше.

Если цены окажутся *выше* тех, что предполагаются, закручивается катастрофическая спираль, и все силы должны быть брошены на то, чтобы ее разорвать. Необходимо выжать все из производительности и разработать новые предложения для поддержки более крупных прогнозов. Затраты должны быть сокращены для того, чтобы получить более низкие оценки. Стресс этого цикла на практике часто порождает наилучшую работу маркетолога и инженера.

При этом возможны забавные колебания. Я вспоминаю машину, счетчик команд которой возникал или убирался из памяти каждые шесть месяцев в течение трехлетнего цикла разработки. В одной фазе требовалось немного больше производительности, поэтому счетчик команд был реализован в транзисторах. В следующей фазе темой стало снижение затрат, поэтому счетчик был реализован в виде ячейки памяти. В еще одном проекте самый лучший менеджер по инженерии, которого я когда-либо видел, часто служил гигантским маховиком — его инерция демпфировала колебания, которые исходили от людей рынка и управленцев.

ДОКУМЕНТЫ ДЛЯ УНИВЕРСИТЕТСКОЙ КАФЕДРЫ

Несмотря на огромные различия в цели и деятельности, аналогичное число схожих документов формирует критический комплект для заведующего кафедрой университета. Почти каждое решение декана, собрания факультета или председателя является уточнением или изменением этих документов:

ЦЕЛЕВЫЕ КРИТЕРИИ.

ОПИСАНИЕ КУРСА.

ТРЕБОВАНИЯ К ДИПЛОМУ.

ИССЛЕДОВАТЕЛЬСКИЕ ПРЕДЛОЖЕНИЯ (ИЛИ ПЛАНЫ, ЕСЛИ ОНИ ФИНАНСИРУЮТСЯ).

РАСПИСАНИЕ (ГРАФИК) ЗАНЯТИЙ И УЧЕБНЫХ ЗАДАНИЙ.

БЮДЖЕТ.

РАСПРЕДЕЛЕНИЕ МЕСТА.

РАСПРЕДЕЛЕНИЕ СОТРУДНИКОВ И АСПИРАНТОВ.

Обратите внимание, что эти компоненты очень похожи на компоненты компьютерного проекта: целевые критерии, спецификации продукта, распределение времени, распределение денег, распределение пространства и распределение людей. Отсутствуют только ценовые документы; здесь эту работу выполняет законодательство. Сходство не случайно — заботами всякой задачи управления являются: что, когда, по какой цене, где и кто.

ДОКУМЕНТЫ ПРОЕКТА

Во многих программных проектах специалисты первым делом проводят совещание, чтобы обсудить структуру, затем начинают писать программы. Однако как бы ни был проект мал, менеджер сразу же мудро приступает к оформлению хотя бы мини-документов, которые послужат ему базой данных. И оказывается, что он нуждается в таких же документах, как и у других менеджеров.

ЧТО: ЦЕЛЕВЫЕ КРИТЕРИИ. В них определяется удовлетворяемая потребность и цели, пожелания, ограничения и приоритеты.

ЧТО: СПЕЦИФИКАЦИИ ПРОДУКТА. Она начинается как техническое предложение и заканчивается как справочное и внутреннее руководство. Скоростные и пространственные характеристики являются критически важной частью.

КОГДА: ГРАФИК РАБОТ.

СКОЛЬКО: БЮДЖЕТ.

ГДЕ: РАСПРЕДЕЛЕНИЕ ПРОСТРАНСТВА.

КТО: ОРГАНИЗАЦИОННАЯ СХЕМА. Она переплетается со спецификацией интерфейса, как предсказывает закон Конвея (Conway): «Организации, проектирующие системы, ограничены дизайном, который

копирует структуру коммуникации в этой организации».¹ Далее Конвей продолжает, указывая на то, что организационная схема будет первоначально отражать первый дизайн системы, который почти наверняка не является правильным. Если дизайн системы должен быть свободен для внесения изменений, то организация должна быть готова к изменениям.

ЗАЧЕМ НУЖНЫ ФОРМАЛЬНЫЕ ДОКУМЕНТЫ?

Во-первых, важно записывать принимаемые решения. Только когда пишешь, становятся видны недочеты и проступают несогласованности. В процессе записывания возникает необходимость принятия сотен мини-решений, и их наличие отличает четкую и ясную политику от расплывчатой.

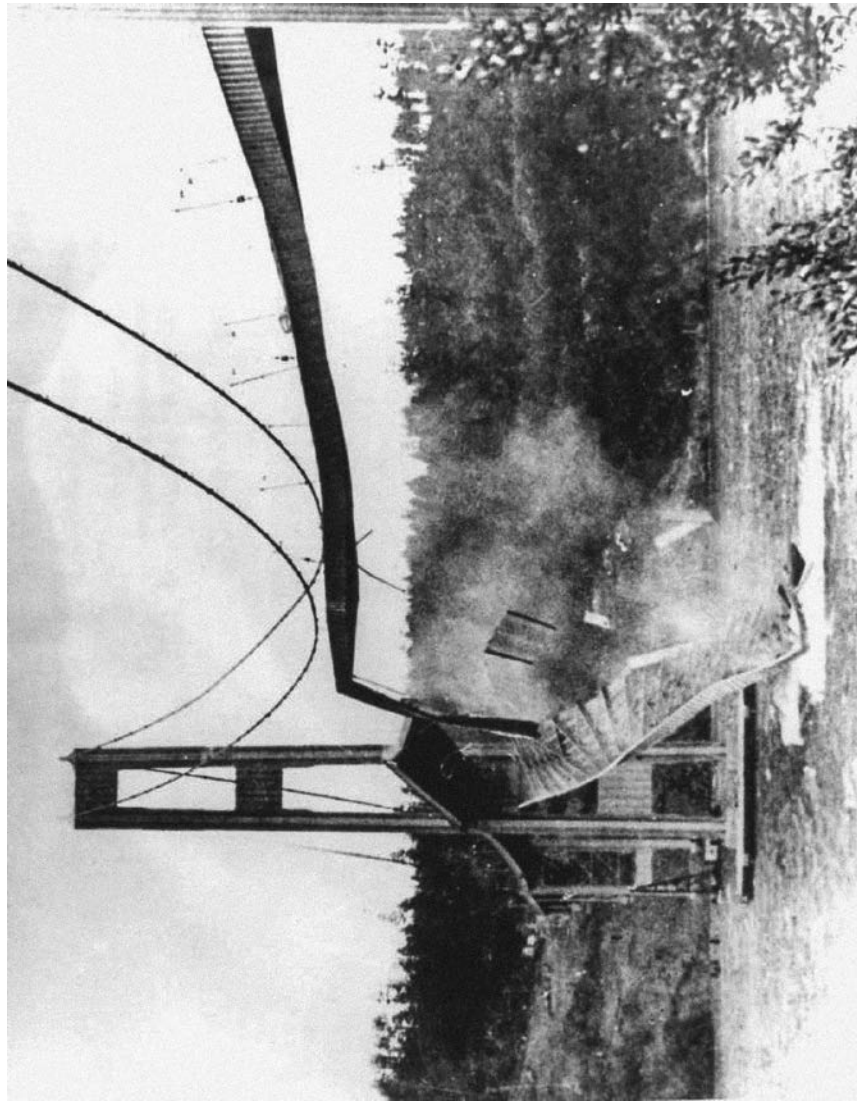
Во-вторых, документы сообщают решения другим людям. Менеджер будет постоянно удивляться, что политика, которую он принимал за общеизвестную, совершенно неизвестна какому-то члену его команды. Поскольку на нем лежит задача удерживать всех в одном направлении, его главной ежедневной работой будет общение, коммуникация, а не принятие решений, и его документы значительно облегчат эту нагрузку.

Наконец, документы предоставляют менеджеру базу данных и контрольный список. Периодически просматривая их, менеджер видит, где находится и какие необходимы изменения или сдвиги в направлении.

Я не разделяю проецируемое продавцами видение «всеобъемлющей информационный управленческой системы», в которой сотрудник вводит запрос в компьютер, а экран дисплея высвечивает ответ. Есть много фундаментальных причин, почему этого никогда не произойдет.

Одна из причин заключается в том, что только небольшая часть, возможно, 20% времени сотрудника исполнительного звена тратится на части работы, которые требуют информации, отсутствующей в его памяти. Остальное — это обмен информацией: выслушивание, отчеты, обучение, увещевание, консультирование, поощрение. Но для той части, которая основана на данных, горстка критически важных документов является жизненно важной, и они удовлетворят почти все потребности.

Задача менеджера — разработать план и затем его реализовать. Но только письменный план является точным и может быть сообщен другим. Такой план состоит из документов о том, что, когда, сколько, где и кто. Этот небольшой комплект важных документов инкапсулирует основную массу работы менеджера. Если их всесторонняя и критическая природа будет подтверждена в самом начале, то менеджер может относиться к ним как к дружественным инструментам, а не как к раздражающей и бесполезной работе. В этом случае он будет задавать свое направление гораздо четче и быстрее.



Крушение Такомаского моста, который был неправильно сконструирован, 1940.
(UPI Photo / Архив Бетмана)

11

ПЛАНИРУЙ ВЫБРОСИТЬ

В этом мире нет ничего постоянного,
кроме непостоянства.

Свифт

Здравый смысл в том, чтобы взять
метод и попробовать его. Если он ока-
зывается неуспешным, то надо честно
признать это и попробовать другой.
Главное — пробовать.

Франклин Д. Рузвельт

ПИЛОТНЫЕ УСТАНОВКИ И ВЕРТИКАЛЬНОЕ МАСШТАБИРОВАНИЕ

Инженеры-химики давно поняли, что процесс, который успешно осуществляется в лабораторных условиях, невозможно реализовать на заводе всего за один шаг. Промежуточный шаг, именуемый *пилотной установкой*, необходим для того, чтобы приобрести опыт в вертикальном масштабировании объемов и эксплуатации в незащищенных средах. Например, лабораторный процесс опреснения воды будет испытан на пилотной установке мощностью 10 тысяч галлонов в день, прежде чем его используют в коммунальной системе водоснабжения мощностью 2 миллиона галлонов в день.

Разработчики программных систем также получили этот урок, но, похоже, он еще не усвоен. Из раза в раз каждый проект сводится к дизайну набора алгоритмов, а затем погружается в сборку программного обеспечения, поставляемого заказчику, по графику, требующему поставки первой вещи, которая будет создана.

В большинстве проектов первая построенная система едва пригодна для пользования. Она может быть слишком медленной, слишком большой, неудобной в использовании или все вместе взятое. Никакой другой альтернативы нет, кроме как начать снова, болезненно, но умнее, и построить переработанную версию, в которой эти проблемы решаются. Отбраковка и редизайн могут быть сделаны одним махом или же по частям. Неважно как, но это будет, что и показывает весь большой системный опыт.¹ Там, где используется новая концепция системы или новая технология, приходится разрабатывать систему, которая потом выбрасывается, потому что даже самое лучшее планирование не является настолько полным, чтобы все получилось правильно с первого раза.

Следовательно, вопрос управления заключается не в том, разрабатывать *или нет* пилотную систему и затем ее выбрасывать. *Вы все равно*

это сделаете. Вопрос только в том, планировать ли заранее разработку системы на выброс или обещать клиентам поставку системы, которую придется выбросить. Если смотреть под этим углом, ответ становится намного проще. С этой точки зрения ответ очевиден. Релиз одноразового продукта клиентам поможет выиграть время, но только за счет агонии пользователя. Побочными эффектами можно назвать и отвлечение внимания строителей, поддерживающих систему в период ее редизайна, и закрепление плохой репутацией для продукта, которую более качественный редизайн едва переживет.

Поэтому планируйте выбросить первую версию — вы все равно это сделаете.

ЕДИНСТВЕННОЕ ПОСТОЯНСТВО — В САМОМ ИЗМЕНЕНИИ

Как только придет осознание того, что пилотная система должна быть построена и выброшена и что редизайн с измененными идеями является неизбежным, ваш опыт обогатится осознанием феномена изменений. Первый шаг — принять факт изменения как образ жизни, а не как досадное и неприятное исключение. Косгроув (Cosgrove) проницательно отметил, что программист поставляет не какой-то материальный продукт, а удовлетворение потребностей пользователя. И реальная потребность, и восприятие этой потребности пользователем будут меняться по мере разработки, тестирования и использования программ.²

Конечно, это также относится к потребностям, удовлетворяемым продуктами аппаратного обеспечения, будь то новые автомобили или новые компьютеры. Но само существование материального объекта служит для сдерживания и квантования запроса пользователей на изменения. В отличие от него как гибкость, так и невидимость программного продукта подвергают его создателей испытанию постоянными изменениями в требованиях.

Я далек от мысли, что все изменения в целевых критериях и требованиях заказчика должны, могут или рекомендованы быть включенными в проект. Очевидно, что должен быть установлен порог, и по мере развития он должен становиться все выше и выше, иначе продукт никогда не появится.

Тем не менее некоторые изменения в целевых критериях неизбежны, и лучше быть к ним готовым, чем уповать на то, что они не наступят. Неизбежны не только изменения в целевых критериях, но и изменения в стратегии и методике развития. Концепция «первый продукт выбросить» сама по себе является просто принятием того факта, что по мере того, как человек учится, он меняет дизайн.

ПЛАНИРУЙ СИСТЕМУ С УЧЕТОМ БУДУЩЕГО ИЗМЕНЕНИЯ

Способы дизайна системы с учетом такого изменения хорошо известны и широко обсуждаются в литературе — возможно, шире обсуждаются, чем практикуются. Они включают тщательную модуляризацию, обширную подпрограмму, точное и полное определение межмодульных интерфейсов и полную документацию по ним. Менее очевидно то, что необходимо применять стандартные последовательности вызовов и табличные методы, используя их везде, где это возможно.

Наиболее важным является использование языка высокого уровня и самодокументированной методики с целью уменьшения ошибок, индуцируемых изменениями. Использование операций времени компиляции для встраивания стандартных объявлений эффективно помогает вносить изменения.

Важным методом является квантование изменений. Каждый продукт должен иметь нумерованные версии, и каждая версия должна иметь свой собственный график и дату заморозки, после чего изменения включают в следующую версию.³

ПЛАНИРУЙ ОРГАНИЗАЦИЮ С УЧЕТОМ БУДУЩЕГО ИЗМЕНЕНИЯ

Косгроув выступает за то, чтобы рассматривать все планы, контрольные точки и графики как предварительные в целях обеспечения изменений. Здесь он преувеличивает — сегодня группы программистов терпят неудачи обычно из-за слишком слабого, а не слишком сильного административного контроля.

Тем не менее он делится интересным наблюдением. По его мнению, нежелание документировать проекты не вызвано исключительно ленью или нехваткой времени. Напротив, оно исходит от нежелания дизайнера брать на себя обязательство защищать решения, которые, как он знает, являются предварительными. «Документируя проект, дизайнер подвергает себя критике со всех сторон, и он должен быть в состоянии защитить все, что пишет. Если организационная структура представляет какую-либо угрозу, то ничего не будет задокументировано до тех пор, пока он не будет полностью защищен».

Структурировать организацию с учетом будущего изменения гораздо сложнее, чем выполнять дизайн системы с учетом такого изменения. Каждый человек должен заниматься заданиями, которые его не ограничивают, с тем чтобы вся рабочая сила была технически гибкой. На крупном проекте менеджеру нужно держать двух или трех ведущих программистов в качестве технической кавалерии, которая прискачет на помощь туда, где сражение наиболее ожесточенное.

Менеджерские структуры также должны изменяться по мере изменения системы. Это означает, что начальник должен уделять большое внимание тому, чтобы его менеджеры и технические работники были взаимозаменяемы настолько, насколько позволяют их таланты.

Эти барьеры носят социальный характер, и с ними беспрерывно необходимо бороться. Во-первых, сами менеджеры часто считают старших сотрудников «слишком ценными», чтобы использовать их в реальном программировании. Далее, чем выше должность, тем выше престиж. Чтобы справиться с этой проблемой, некоторые лаборатории, такие как *Bell Labs*, отменяют все названия должностей. Каждый профессиональный сотрудник является «членом технической команды». Другие, такие как IBM, поддерживают двойную служебную лестницу, как показано на рис. 11.1. Соответствующие ступени теоретически являются эквивалентными.



Рис. 11.1. Двойная служебная лестница, принятая в IBM

Для ступеней лестницы легко устанавливаются соответствующие шкалы окладов. Гораздо труднее придать им соответствующий престиж. Офисы должны быть одинакового размера и назначения. Секретарские и другие вспомогательные службы должны соответствовать друг другу. Переназначение с технической лестницы на соответствующий уровень на менеджерской лестнице никогда не должно сопровождаться повышением, и оно всегда должно объявляться как «перевод», а не как «повышение». Обратное переназначение всегда должно сопровождаться повышением оплаты — необходимая компенсация для преодоления культурных стереотипов.

Менеджеров необходимо направлять на технические курсы повышения квалификации, старших технических сотрудников — на управленческие курсы повышения квалификации. Целевые критерии проекта, ход выполнения и проблемы менеджмента должны доводиться до сведения всего корпуса старших сотрудников.

Всякий раз, когда позволяют ресурсы, старшие сотрудники должны быть технически и эмоционально готовы управлять группами или получать удовольствие от создания программ своими руками. Занятие этим, конечно, выливается в большой объем работы, но оно, безусловно, стоит того!

Вся идея организации команд по принципу хирургических является радикальным решением этой проблемы. Она помогает старшему сотруднику не чувствовать себя «униженным», когда он создает программы, и пытается устранить социальные препятствия, которые лишают созидательной творческой радости.

Более того, эта структура предназначена для минимизации числа интерфейсов. Как таковая, она делает систему максимально легкой в изменении и относительно упрощает переназначение всей хирургической команды на другую программистскую работу, когда организационные изменения становятся необходимыми. Это действительно является долгосрочным ответом на задачу гибкой организации.

ДВА ШАГА ВПЕРЕД И ОДИН ШАГ НАЗАД

Программа не перестает меняться, даже когда она уже используется клиентом. Изменения после поставки называются сопровождением программного обеспечения, но этот процесс принципиально отличается от сопровождения аппаратного обеспечения.

Сопровождение компьютерной системы предусматривает три вида работ: замену вышедших из строя компонентов, очистку и смазку,

а также внесение инженерных изменений, устраняющих конструктивные ошибки. (Большинство, но не все инженерные изменения исправляют ошибки не в архитектуре, а в реализации и поэтому невидимы для пользователя.)

Сопровождение программного обеспечения не включает в себя очистку, смазку или ремонт изношенных деталей. Оно в основном состоит из изменений, которые устраняют ошибки дизайна. Гораздо чаще, чем это бывает с аппаратным обеспечением, эти изменения включают в себя добавочные функции. Обычно они видны пользователю.

Суммарная стоимость сопровождения широко используемой программы, как правило, составляет 40 % или более от стоимости ее разработки. Удивительно, но на эту стоимость сильно влияет число пользователей. Чем больше пользователей, тем больше ошибок они находят.

Бетти Кэмпбелл (Betty Campbell) из лаборатории ядерной физики Массачусетского технологического института указывает на интересный цикл в жизни конкретного релиза программы. Это показано на рис. 11.2. Первоначально старые ошибки, найденные и решенные в предыдущих релизах, имеют тенденцию появляться в новом релизе. Как оказалось, новые функции нового релиза имеют ошибки. Эти недостатки исправляются, и все идет хорошо в течение нескольких месяцев. Затем частота встречаемости ошибок снова начинает расти. Мисс Кэмпбелл считает, что это связано с приходом пользователей на новое плато сложности, где они начинают в полной мере использовать новые возможности релиза. Такая интенсивная отработка затем выявляет более тонкие ошибки в новых функциях.

Фундаментальная проблема с сопровождением программного обеспечения заключается в том, что исправление ошибки имеет существенный (20–50 %) шанс привнести еще одну ошибку. Таким

образом, весь процесс представляет собой два шага вперед и один шаг назад.



Рис. 11.2. Частота ошибок как функция от возраста релиза

Почему ошибки не устраняются более аккуратно? Во-первых, даже малозаметная ошибка проявляется как некий локальный сбой. На самом деле он часто имеет общесистемные последствия, обычно неочевидные. Любая попытка исправить его с минимальными усилиями приведет к восстановлению локального и очевидного, но если структура не будет чистой или документация очень точной, то далеко идущие последствия ремонтно-восстановительных работ не будут замечены. Во-вторых, ремонтник, как правило, является не тем человеком, который написал код, и часто он является младшим программистом или стажером.

Вследствие появления новых ошибок сопровождение программного обеспечения требует гораздо большего тестирования системы в расчете на каждую написанную инструкцию, чем любое другое

программирование. Теоретически после каждого исправления необходимо прогнать весь банк тестовых случаев, ранее выполненных относительно системы, с целью удостовериться, что она не была повреждена неясным образом. На практике такое *регрессионное тестирование* действительно должно быть приближено к теоретическому идеалу, и это очень дорого.

Очевидно, методы разработки программ, позволяющие исключить или, по крайней мере, выявить побочные эффекты, могут резко снизить стоимость сопровождения, как и методы разработки проектов меньшим числом людей и с меньшим числом интерфейсов — а значит, и с меньшим числом ошибок.

ШАГ ВПЕРЕД И ОДИН НАЗАД

Леман (Lehman) и Белادي (Belady) изучили историю последовательных релизов крупной операционной системы.⁴ Они обнаружили, что общее число модулей увеличивается линейно вместе с номером релиза, но при этом число затронутых модулей увеличивается экспоненциально вместе с номером релиза. Все исправления тяготеют к разрушению структуры, увеличению энтропии и дезорганизации системы. Все меньше и меньше усилий тратится на исправление первоначальных недостатков дизайна; все больше и больше тратится на исправление недостатков, внесенных более ранними исправлениями. С течением времени система становится все менее упорядоченной. Рано или поздно исправление ошибок перестает приносить плоды. Каждый шаг вперед сопровождается шагом назад. Хотя в принципе эта система может использоваться вечно, она изнашивается как основа для продвижения вперед. Кроме того, меняются машины, конфигурации и требования пользователей, поэтому система фактически не может использоваться вечно. Становится необходимым совершенно новый проект с нуля.

Таким образом, для систем программирования Белادي и Леман приходят от статистической механической модели к более общему заключению, подкрепляемому всеобщим опытом. Как сказал Паскаль: «Поначалу все всегда складывается наилучшим образом». К. С. Льюис (C. S. Lewis) высказал эту мысль еще пронизательнее:

В этом ключ к истории. Расходуется колоссальная энергия, создаются цивилизации, создаются прекрасные учреждения, но каждый раз что-то идет не так. Какая-то фатальная ошибка всегда приводит эгоистичных и жестоких людей к вершине, а затем все это снова скатывается в нищету и разорение. На самом деле машина заглохла. Кажется, что она в полном порядке — запускается, пробегает несколько ярдов, а затем ломается.⁵

Создание системных программ — это процесс уменьшения энтропии, следовательно, в сущности, он является метастабильным. Сопровождение программного обеспечения — это процесс увеличения энтропии, и даже самое искусное его исполнение лишь оттягивает момент неисправимого устаревания системы.



А. Пизано. «Скульптор», из Campanile di Santa Maria del Fiore, Флоренция, ок. 1335

12

ЗАТОЧЕННЫЕ ИНСТРУМЕНТЫ

Хорошего работника видно
по его инструментам.

Пословица

Даже в наш высокотехнологичный век многие программные проекты по-прежнему работают как токарные мастерские, когда речь идет об инструментах. Каждый мастер тщательно охраняет свой личный набор, собранный за всю жизнь, — видимые свидетельства личных навыков. Точно так же программист хранит маленькие редакторы, сортировщики, двоичные дампы, дисковые утилиты и т. д., припрятанные в его папке.

Однако для программного проекта такой подход является несуразным. Во-первых, существенной проблемой является коммуникация, а индивидуализированные инструменты скорее препятствуют, чем помогают коммуникации. Во-вторых, технология меняется, когда меняются машины или рабочий язык, поэтому срок службы инструмента короток. Наконец, очевидно, что гораздо эффективнее иметь общую разработку и сопровождение универсальных инструментов программирования.

Однако универсальных инструментов недостаточно. Специализированные потребности и личные предпочтения диктуют необходимость в специализированных инструментах, поэтому при обсуждении состава команды программистов я предлагал иметь в бригаде одного инструментальщика. Этот человек владеет всеми обычными инструментами и может обучить своего клиента-босса их использовать. Он также разрабатывает специализированные инструменты, необходимые руководству.

Менеджер проекта, таким образом, должен определять философию и выделять ресурсы для разработки общих инструментов. В то же время он должен признавать необходимость специализированных инструментов и поощрять стремление его рабочих команд разрабатывать собственные инструменты. Это искушение является коварным. Кажется, что если бы разрозненные инструментальщики были собраны вместе для усиления команды по общим инструментам, то это привело бы к большей эффективности. Но это не так.

Что это за инструменты, о которых менеджер должен задуматься, спланировать и организовать их разработку? Во-первых, *компьютерное обеспечение*. Оно обуславливает необходимость наличия машин, и должна быть принята философия определения графика их задействования. Предполагается наличие *операционной системы* и философии служб. Предполагается наличие *языка* и установление языковой политики. Во-вторых, также есть *утилиты*, *вспомогательные средства отладки*, *генераторы тестовых случаев* и *система обработки текста* для документации. Давайте рассмотрим их один за другим.¹

ЦЕЛЕВЫЕ МАШИНЫ

Машинную поддержку полезно разделить на *целевую* (target) *машину* и *несущие* (vehicle) *машины*. Целевая машина — это машина, для которой пишется программное обеспечение и на которой оно должно быть протестировано. Несущие машины — это машины, которые используются в разработке системы. Если строится новая операционная система для старой машины, то такая машина может служить не только в качестве целевой, но и в качестве несущей.

Какого рода целевое обеспечение требуется? Команды разработки, строящие новые супервизоры или другое системное программное обеспечение, разумеется, будут нуждаться в собственных машинах. Такие системы будут нуждаться в операторах и в одном-двух системных программистах, которые осуществляют стандартную поддержку на текущей и работоспособной машине.

Если нужна отдельная машина, то это довольно своеобразная вещь — она не должна обязательно быть быстрой, но ей нужен по крайней мере миллион байт основного хранилища, 100 миллионов байт онлайн-диска и терминалы. Необходимы только буквенно-цифровые терминалы, но они должны быть гораздо

быстрее, чем скорость 15 символов в секунду, которая характерна для пишущих машинок. Наличие большой памяти значительно способствует продуктивности, позволяя заняться разбиением на оверлеи и минимизацией размера после тестирования функций.

Отладочная машина либо ее ПО также должны быть инструментированы для того, чтобы подсчет количеств и измерения всех видов параметров программы могли выполняться автоматически во время отладки. Шаблоны использования памяти, например, являются мощной диагностикой причин странного логического поведения или неожиданно низкой производительности.

ОПРЕДЕЛЕНИЕ ГРАФИКА ИСПОЛЬЗОВАНИЯ МАШИННОГО ВРЕМЕНИ. Когда целевая машина является новой, как и при создании первой операционной системы, машинного времени не хватает, и определение графика его распределения является главной проблемой. Потребность во времени для целевой машины имеет своеобразную кривую роста. В разработке OS/360 у нас были хорошие симуляторы системы System/360 и другие носители. Из предыдущего опыта мы спроецировали, сколько часов работы S/360 нам понадобится, и начали приобретать у производителя ранние машины. Но месяц за месяцем они стояли без дела. Затем все 16 систем были полностью загружены, и встала проблема нормирования. Динамика их задействования выглядела примерно как на рис. 12.1. Все начали отлаживать свои первые компоненты одновременно, и после этого большинство команды постоянно что-то отлаживало.

Мы централизовали все наши машины и библиотеку магнитных лент и создали для обслуживания их работы профессиональную и опытную команду машинного зала. С целью максимизации скудного времени S/360 мы выполняли все отладочные прогоны пакетно на любой системе, которая была свободной и подходящей. Мы проводили четыре попытки в день (с оборотом в два с половиной часа) и запрашивали четырехчасовой оборот. Вспомогательный модуль 1401 с терминалами использовался для определения

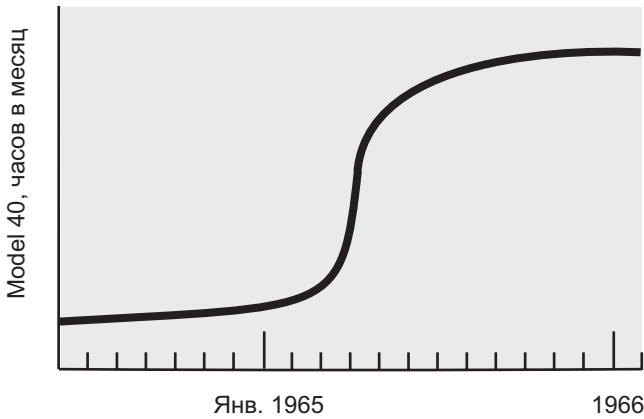


Рис. 12.1. Динамика использования целевых машин

графика прогонов, отслеживания тысяч заданий и мониторинга обратного времени.

Но вся эта организация была явно избыточна. После нескольких месяцев медленных оборотов, взаимных обвинений и прочих мучений мы перешли к выделению машинного времени значительными блоками.

Например, всей сортировочной команде из 15 человек система предоставлялась для 4–6-часового блока. И они уже сами на нем распределяли время между собой. Если их система стояла без дела, то ни один посторонний не мог бы ею воспользоваться.

Данный способ распределения времени и определения графика, как выясняется, был оптимальным. Хотя задействование машин, возможно, было немного ниже (и часто это было не так), продуктивность была намного выше. Для каждого члена такой команды 10 попыток в шестичасовом блоке являются гораздо более продуктивными, чем 10 попыток с интервалом в 3 часа, потому что постоянная концентрация сокращает время на размышление.

После такого спринта команде обычно требовался день-два на то, чтобы разобраться с бумагами, прежде чем запрашивать еще один блок. Часто не больше трех программистов способны плодотворно делить и распределять блок времени между собой. Этот способ использования целевой машины при отладке новой операционной системы, по-видимому, является самым лучшим.

Так было всегда на практике, но никогда — в теории. Отладка системы всегда становилась уделом ночной смены, как астрономия. Двадцать лет назад на 701-й я был посвящен в продуктивную неформальность предрассветных часов, когда начальники машинного зала крепко спят дома, а операторы нарушают правила. Минуты три поколения машин, технологии полностью изменились, появились операционные системы, и все же этот предпочтительный метод работы не изменился. Он продолжает жить, потому что является наиболее продуктивным. Пришло время признать его продуктивность и открыто внедрять плодотворную практику.

НЕСУЩИЕ МАШИНЫ И СЛУЖБЫ ДАННЫХ

ЭМУЛЯТОРЫ. Если целевой компьютер является новым, то для него требуется логический эмулятор. Это дает аппарат для отладки задолго до того, как целевая машина будет в наличии. Не менее важно, что он дает доступ к надежному отладочному средству даже после того, как у вас есть в наличии целевая машина.

Надежный вовсе не означает *точный*. Эмулятор наверняка не сможет в каком-то отношении быть верной и точной имплементацией архитектуры новой машины. Но изо дня в день это будет одна и та же имплементация, чего не скажешь о новой аппаратной части.

В наши дни мы привыкли к тому, что компьютерное обеспечение работает правильно почти все время. В случае, если прикладной программист не видит, что от прогона к идентичному прогону

система ведет себя противоречиво, то ему рекомендуется искать ошибки в своем коде, а не в своем двигателе.

Этот опыт, однако, является плохим тренировочным примером в программировании поддержки новой машины. Построенное в лабораторных условиях, предварительное или раннее аппаратное обеспечение *не* надежно, *не* работает так, как определено, и *не* остается неизменным изо дня в день. По мере обнаружения ошибок инженерные изменения вносятся во все машинные копии, включая копии группы по программированию. Эта смещающаяся база является довольно плохой. Хуже — сбои в аппаратном обеспечении, которые обычно происходят нерегулярно. Но хуже всего неопределенность, потому что она лишает стимула усердно копаться в своем коде в поисках ошибки — ее может там вообще не быть. Таким образом, надежный симулятор на довольно устаревшей несущей машине сохраняет свою полезность гораздо дольше, чем можно было бы ожидать.

КОМПИЛЯТОРНАЯ И АССЕМБЛЕРНАЯ НЕСУЩИЕ МАШИНЫ. По тем же причинам требуются компиляторы и ассемблеры, работающие на надежных несущих машинах, но компилирующие объектный код для целевой системы. Его затем можно начать отлаживать на симуляторе.

С помощью программирования на языке высокого уровня можно проделывать большую часть отладки путем компиляции и тестирования объектного кода на несущей машине и только потом вообще начинать тестировать код целевой машины. Это обеспечивает эффективность прямого исполнения вместо симулирования в сочетании с надежностью стабильной машины.

БИБЛИОТЕКИ ПРОГРАММ И ОТЧЕТНОСТЬ. Весьма успешное и важное использование несущей машины в работе по созданию OS/360 заключалось в сопровождении библиотек программ. Система, разработанная под руководством У. Р. Кроули (W. R. Crowley), имела две

подключенные машины 7010 с общей дисковой базой данных. На 7010 также поддерживался ассемблер S/360. Весь протестированный или находящийся в стадии тестирования код хранился в этой библиотеке, то есть и исходный код, и собранные подгружаемые модули. Библиотека была фактически разделена на подбиблиотеки с разными правилами доступа.

Сначала у каждой группы или программиста была своя область — игровая площадка, где они хранили копии своих программ, свои тестовые случаи и строительные леса, необходимые для тестирования компонентов. На этой *площадке для игр* не было никаких ограничений на действия с собственными программами.

Когда у члена команды появлялся свой компонент, готовый для интеграции в более крупную часть, он передавал копию менеджеру этой более крупной системы, который помещал эту копию в *подбиблиотеку системной интеграции*. Теперь автор уже не мог ее изменить, кроме как с разрешения менеджера по интеграции. По мере того как система объединялась, последняя проходила всевозможные системные тесты, выявляя ошибки и совершенствуясь.

Время от времени версия системы становилась готовой для более широкого использования. Затем она переводилась в *подбиблиотеку текущей версии*. Эта копия была неприкосновенной, ее брали только для того, чтобы исправить фатальные ошибки. Она была доступна для использования в интеграции и тестировании всех новых версий модулей. Каталог программ в 7010 отслеживал каждую версию каждого модуля, его состояние, местонахождение и изменения.

Здесь важны два понятия. Во-первых, это *контроль*, идея копий программ, принадлежащих менеджером, являющимся единственными, кто может санкционировать их изменение. Во-вторых, это *формальное разделение и продвижение* от игровой площадки к интеграции и далее к релизу.

На мой взгляд, это было одним из лучших достижений в разработке OS/360. Эта часть технологии менеджмента, по-видимому, была независимо разработана на нескольких крупных проектах, включая проекты Bell Labs, ICL и Кембриджского университета.² Она применима как к документации, так и к программам. Эта технология является незаменимой.

ИНСТРУМЕНТЫ ДЛЯ ПРОГРАММИРОВАНИЯ. По мере появления новых отладочных методов старые теряют значение, но не исчезают. Следовательно, нужны дампы, редакторы исходных файлов, дампы мгновенного состояния, даже трассировки.

Точно так же необходим полный комплект утилит для переноса данных с перфокарт на диски, создания ленточных копий, печати файлов, изменения каталогов. Если кто-то заказывает инструментальщика проекта в начале процесса, то единожды разработанные инструменты могут использоваться по мере необходимости.

СИСТЕМА ДОКУМЕНТИРОВАНИЯ. Среди всех инструментов тот, который экономит больше всего труда, вполне может являться компьютеризированной системой редактирования текста, работающей на надежной несущей машине. У нас был такой очень удобный инструмент, разработанный Дж. У. Франклином (J. W. Franklin). Без него, по моим ощущениям, справочные руководства по OS/360 вышли бы намного позже и были бы более запутанными. Некоторые утверждают, что шестифутовая полка справочных руководств по OS/360 представляет собой словесное недержание, что сама их объемность рождает новый вид непостижимости. И в этом есть доля правды.

Но у меня есть два возражения. Во-первых, хотя объем документации по OS/360 угнетает, его план чтения тщательно продуман; если изучать материал выборочно, можно пропустить большую часть документов и тем самым сэкономить много времени. Документацию по OS/360 следует рассматривать как библиотеку или энциклопедию, а не как набор обязательных текстов.

Во-вторых, это гораздо предпочтительнее серьезного недостаточного документирования, которое характерно для большинства систем программирования. Я охотно соглашусь, что в некоторых местах изложение может быть значительно улучшено и что результатом более качественного изложения будет сокращение объема. Некоторые части (например, *концепции и средства обеспечения*) сейчас очень хорошо написаны.

ЭМУЛЯТОР ПРОИЗВОДИТЕЛЬНОСТИ. Его лучше иметь. Разрабатывайте его «снаружи внутрь», как мы обсудим в следующей главе. Используйте один и тот же нисходящий дизайн для эмулятора производительности, логического эмулятора и продукта. Начните как можно раньше. Прислушайтесь к тому, что он вам скажет.

ЯЗЫК ВЫСОКОГО УРОВНЯ И ИНТЕРАКТИВНОЕ ПРОГРАММИРОВАНИЕ

Самыми важными для системного программирования сегодня являются именно те два инструмента, которые не использовались в разработке OS/360 почти 10 лет назад. Они до сих пор широко не используются, но все свидетельства указывают на их мощь и применимость. Это (1) язык высокого уровня и (2) интерактивное программирование. Убежден, что только инертность и лень препятствуют всеобщему принятию этих инструментов; технические трудности больше не являются допустимым оправданием.

ЯЗЫК ВЫСОКОГО УРОВНЯ. Основными причинами использования языка высокого уровня являются продуктивность и скорость отладки. Мы уже говорили о продуктивности (глава 8). Существует не так много численных доказательств, но то, что есть, предполагает кратное улучшение, а не только на инкрементный процент.

Улучшение отладки происходит из-за того, что ошибок становится меньше и их легче отыскать. Их меньше, потому что человек избегает целого уровня подверженности ошибкам, уровня, на котором он делает не только синтаксические, но и семантические ошибки, такие как неправильное использование регистров. Ошибки легче найти, потому что диагностика компилятора помогает их отыскать и, что важнее, потому что очень легко вставлять отладочные снимки.

Для меня эти причины продуктивности и отладки являются вескими. Я не могу легко представить себе систему программирования, которую построил бы на ассемблере.

Ну а как же классические возражения против такого инструмента? Их три: они не позволяют мне делать то, что я хочу. Объектный код слишком велик. Объектный код слишком медленный.

Что касается функциональности, то я считаю, что это возражение больше не имеет силы. Все свидетельствует в пользу того, что можно делать то, что хочется, потрудившись найти способ, но иногда для этого приходится изловчиться.^{3,4}

Что касается пространства, то новые оптимизирующие компиляторы начинают показывать весьма удовлетворительные результаты, и это улучшение будет продолжаться.

Что касается скорости, то оптимизирующие компиляторы теперь производят некоторый код, который быстрее, чем рукописный код большинства программистов, кроме того, обычно можно решить проблемы скорости, заменив от 1 до 5 % сгенерированной компилятором программы рукописной вставкой после того, как первая полностью отлажена.⁵

Какой язык высокого уровня следует использовать для системного программирования? Единственным разумным кандидатом

на сегодняшний день является PL/1.⁶ Он имеет очень полный набор функций, соответствует операционной среде; имеется целый ряд компиляторов с разными особенностями — интерактивных, быстрых, с улучшенной диагностикой, с высокой степенью оптимизации. Лично я быстрее разрабатываю алгоритмы с помощью APL, а затем перевожу их в PL/I для соответствия системному окружению.

ИНТЕРАКТИВНОЕ ПРОГРАММИРОВАНИЕ. Одним из обоснований проекта Multics из MIT была его полезность для сборки систем программирования. Multics (а вслед за ним и IBM TSS) отличается по своей концепции от других интерактивных вычислительных систем именно тем, что требуется для системного программирования: многоуровневым обменом данными и программами, обширным управлением библиотеками и возможностями для совместной работы пользователей терминалов. Убежден, что во многих приложениях интерактивные системы никогда не вытеснят пакетные системы. Но думаю, что команда Multics привела самые убедительные доводы в ее пользу именно в применении к системному программированию.

Пока еще не существует большого количества доказательств истинной плодотворности таких, казалось бы, мощных инструментов. Существует широкое признание того, что отладка является трудной и медленной частью системного программирования, а медленный оборот является проклятием отладки. Так что логика интерактивного программирования кажется неумолимой.⁷

Более того, об этом свидетельствуют многие, кто строил крупные системы или части систем таким образом. Единственные доступные мне данные относительно влияния на программирование больших систем были сообщены Джоном Харром из Bell Labs. Они показаны на рис. 12.2. Эти цифры относятся к написанию, сборке и отладке программ. Первая программа — это в основном управляющая программа, остальные три — трансляторы, редакторы и т. п. Данные

Программа	Размер	Пакетная (П) или диалоговая (Д)	Операторов на человека в год
Код ESS	800 000	П	500–1000
Поддержка ESS 7094	120 000	П	2100–3400
Поддержка ESS 360	32 000	Д	8000
Поддержка ESS 360	8300	П	4000

Рис. 12.2. Сравнительная продуктивность в условиях пакетного и диалогового программирования

Харра свидетельствуют о том, что интерактивное средство обеспечения по крайней мере удваивает продуктивность в системном программировании.⁸

Эффективное использование большинства интерактивных инструментов требует, чтобы работа выполнялась на языке высокого уровня, поскольку телетайпный и печатный терминалы не могут использоваться для отладки путем сброса памяти. С помощью языка высокого уровня исходный код можно легко редактировать и легко делать выборочные распечатки. Вместе они действительно составляют пару заточенных инструментов.



disneyscreencomps.com

© The Walt Disney Company

13

ЦЕЛОЕ И ЧАСТИ

Я духов вызывать могу из бездны.
И я могу, и каждый может.
Вопрос лишь, явятся ль на зов они?*

Шекспир. Король Генрих IV

* Перевод Е. Бируковой. — *Примеч. ред.*

Среди современных волшебников, как и встарь, встречаются хвастуны: «Я могу писать программы, которые управляют воздушным движением, перехватывают баллистические ракеты, сверяют банковские счета, контролируют производственные линии». На что приходит ответ: «Я тоже могу, и любой человек может, но будут ли они работать, когда вы их напишете?»

Как написать работающую программу? Как ее протестировать? И как интегрировать протестированный набор компонентных программ в протестированную и надежную систему? Мы уже касались этих методов здесь и там; давайте теперь рассмотрим их несколько системно.

ДИЗАЙН, ИСКЛЮЧАЮЩИЙ ОШИБКИ

ОБЕСПЕЧЕНИЕ ДЕФЕКТСТОЙКОСТИ ОПРЕДЕЛЕНИЯ. Наиболее пагубными и едва уловимыми дефектами являются системные ошибки, возникающие из-за несовпадающих допущений, принимаемых авторами разных компонентов. Подход к концептуальной целостности, рассмотренный выше в главах 4, 5 и 6, затрагивает эти проблемы непосредственно. Одним словом, концептуальная целостность продукта не только упрощает его использование, но и облегчает его разработку и делает менее подверженным ошибкам.

Такую же работу выполняют детальные, кропотливые архитектурные усилия, вытекающие из указанного подхода. В. А. Высоцкий из проекта Safeguard, выполнявшегося в *Bell Telephone Laboratories*, говорит, что «важнейшая работа — дать продукту определение. Многие, очень многие неудачи касаются именно тех аспектов, которые никогда не были точно определены».¹ Тщательное определение функций, тщательная спецификация и дисциплинированное изгнание излишеств функциональности и полетов технической

мысли — все это уменьшает число системных ошибок, которые должны быть найдены.

ТЕСТИРОВАНИЕ СПЕЦИФИКАЦИИ. Задолго до того, как появится какой-либо код, спецификация должна быть передана группе внешнего тестирования для тщательного изучения на предмет полноты и ясности. Как говорит Высоцкий, сами разработчики не могут этого сделать: «Они не могут признаться, что не понимают ее, они будут счастливо прокладывать свой путь через пропущенные и темные места».

НИСХОДЯЩИЙ ДИЗАЙН. В подробной статье 1971 года Никлаус Вирт (Niklaus Wirth) формализовал процедуру дизайна, которая в течение многих лет использовалась лучшими программистами.² Более того, его понятия, хотя и сформулированные для дизайна программ, полностью применимы к дизайну комплексных систем программ. Деление процесса сборки системы на архитектуру, имплементацию и реализацию является воплощением этих понятий; более того, каждая архитектура, имплементация и реализация могут быть наилучшим образом выполняться нисходящими методами.

Вкратце, процедура Вирта состоит в том, чтобы определить дизайн как последовательность *уточняющих шагов*. Делается грубый набросок определения работы и грубый метод ее решения, который достигает принципиального результата. Затем указанное определение изучается более внимательно с целью увидеть, как результат отличается от того, что требуется, а крупные шаги решения разбиваются на более мелкие. Каждое уточнение в определении задачи становится уточнением в алгоритме решения, и каждое может сопровождаться уточнением в представлении данных.

Из этого процесса выявляют *модули* решения или модули данных, дальнейшее уточнение которых может происходить независимо от другой работы. Степень этой модульности определяет адаптивность и изменчивость программы.

Вирт выступает за использование как можно более высокоуровневой нотации на каждом этапе, раскрывая концепции и скрывая детали до тех пор, пока не станет необходимым дальнейшее уточнение.

Хороший нисходящий дизайн позволяет избежать ошибок несколькими путями. Во-первых, ясность структуры и представления облегчает точное изложение требований и функций модулей. Во-вторых, разделение и независимость модулей позволяют избежать системных ошибок. В-третьих, подавление деталей делает недостатки в структуре более очевидными. В-четвертых, дизайн может быть протестирован на каждом шаге его уточнения, поэтому тестирование можно начать раньше и сосредоточиться на надлежащем уровне детализации на каждом шаге.

Процесс поэтапного уточнения не означает, что никогда не нужно возвращаться назад, отбрасывать верхний уровень и начинать все сначала, когда он сталкивается с какой-то неожиданно затруднительной деталью. Действительно, такое случается часто. Но при этом гораздо легче понять, когда и почему нужно отбросить проект и начать все сначала. Многие плохие системы возникают из попытки спасти плохой базовый дизайн и залатать косметическими заплатками. Нисходящий дизайн уменьшает такой соблазн.

Убежден, что нисходящий дизайн является самой важной новой формализацией программирования последнего десятилетия.

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ. Еще один важный набор новых идей для бездефектного дизайна в программах во многом вытекает из работ Дейкстры³ и построен на теоретической структуре Бёма (Boehm) и Якопини (Jacopini).⁴

По сути, указанный подход заключается в дизайне программ, управляющие структуры которых состоят только из циклов, заданных инструкцией языка, такой как `DO WHILE`, и условных частей, разделенных на группы инструкций, помеченных скобками и обус-

ловленных конструкцией `IF ... THEN... ELSE`. Бом и Якопини показывают, что эти структуры являются теоретически достаточными; Дейкстра утверждает, что альтернативное, неограниченное ветвление через `GO TO` производит структуры, которые подвержены логическим ошибкам.

В основе, несомненно, лежат здравые мысли. Было высказано много критических замечаний, и дополнительные управляющие структуры, такие как множественное ветвление (так называемая инструкция `CASE`), для различения среди непредвиденных обстоятельствах и урегулирования аварийных ситуаций (`GO TO ABNORMAL END`) оказались очень удобными. Более того, некоторые критики заняли очень доктринерскую позицию в отношении того, чтобы избегать всех переходов `GO TO`, и это кажется уже чрезмерным.

Значимым моментом и жизненно важным для создания бездефектных программ является то, что мы хотим думать об управляющих структурах системы как об управляющих структурах, а не как об отдельных инструкциях ветвления. Такой образ мышления является важным шагом вперед.

ОТЛАДКА КОМПОНЕНТОВ

За последние 20 лет процедуры отладки программ для устранения ошибок прошли через большой цикл, и в некотором смысле они вернулись к тому, с чего начинали. Указанный цикл прошел четыре шага, и любопытно проследить их и увидеть мотивацию для каждого.

ОТЛАДКА НА МЕСТЕ. Ранние машины имели относительно плохое оборудование ввода-вывода и длинные задержки ввода-вывода. Как правило, машина считывала и писала на бумажную или магнитную ленту, и автономные средства обеспечения исполь-

зовались для подготовки ленты и печати. Это делало ленточный ввод-вывод невыносимо неудобным для отладки, поэтому вместо него использовалась консоль. Таким образом, отладка организовывалась так, чтобы обеспечить как можно больше тестов за машинный сеанс.

Программист составлял тщательный дизайн своей процедуры отладки, планируя, где остановиться, какие области памяти проверять, что там найти и что делать, если он не найдет. Это дотошное программирование отладочной логики само по себе могло занимать половину времени от написания отлаживаемой программы.

Смертный грех состоял в том, чтобы смело нажимать кнопку START, не разбив программу на тестовые секции с запланированными остановами.

ДАМПЫ ПАМЯТИ. Отладка на месте была очень эффективной. За двухчасовой сеанс можно было сделать около дюжины снимков. Но компьютеры были очень редки и очень дороги, и мысль о том, что все это машинное время будет потрачено впустую, была ужасающей.

Поэтому когда высокоскоростные принтеры были подключены к сети, методика изменилась. Теперь программа выполнялась до тех пор, пока проверка не нарушалась, а затем делался дамп всей памяти. Потом начиналась кропотливая работа за письменным столом, в которой изучалось содержимое каждой ячейки памяти. Время за письменным столом не сильно отличалось от времени отладки на месте; но оно происходило после выполнения теста, при расшифровке, а не до, при планировании. У любого конкретного пользователя отладка занимала гораздо больше времени, поскольку тестовые снимки зависели от пакетного оборотного времени. Вся процедура, однако, была составлена так, чтобы минимизировать потребление компьютерного времени и обслуживать как можно больше программистов.

СНИМКИ. Машины, на которых было разработано даммирование (выгрузка дампа) памяти, имели память емкостью 2000–4000 слов, или 8–16 Кбайт. Но размеры памяти росли семимильными шагами, и полная выгрузка дампов стала непрактичной. Поэтому люди разработали методы выборочных дампов, выборочной трассировки и вставки снимков в программы. TESTRAN в OS/360 является вершиной в линейке подобных средств, позволяя вставлять снимки в программу без повторной сборки или перекompиляции.

ИНТЕРАКТИВНАЯ ОТЛАДКА. В 1959 году Кодд (Codd) и его коллеги⁵ и Стрейчи⁶, каждый в отдельности, сообщили о работе, целью которой была отладка в режиме разделения времени, позволяющая одновременно достичь мгновенной оборачиваемости отладки в активном режиме и эффективно использовать машинное время, как при пакетной обработке заданий. Компьютер имел несколько программ в памяти, готовых к исполнению. Терминал, управляемый только программой, ассоциировался с каждой отлаживаемой программой. Отладка осуществлялась под контролем управляющей программы. Когда программист на терминале останавливал свою программу с целью проверки хода выполнения или внесения изменений, супервизор запускал еще одну программу, таким образом оставляя машины все время занятыми.

У Кодда была разработана система мультипрограммирования, но акцент был сделан на повышение пропускной способности за счет эффективного использования ввода-вывода, а интерактивная отладка не была имплементирована. Идеи Стрейчи были усовершенствованы и имплементированы в 1963 году Корбатом и его коллегами из MIT⁷ в экспериментальной системе для 7090. Эта разработка привела к появлению MULTICS, TSS и других современных систем с совместным использованием времени.

Главными ощущаемыми пользователем различиями между отладкой в активном режиме, как она осуществлялась ранее, и сегодняшней интерактивной отладкой являются возможности, полученные

в результате присутствия программы-супервизора и связанных с ней интерпретаторов языков программирования. Можно программировать и производить отладку на языках высокого уровня. Эффективные средства редактирования позволяют легко делать изменения и моментальные снимки.

Возврат к способности мгновенного оборота отладки на месте еще не привел к возврату к предварительному планированию сеансов отладки. В каком-то смысле такое предпланирование уже не является таким необходимым, как раньше, поскольку машинное время не пропадает даром, пока человек сидит и думает.

Тем не менее интересные экспериментальные результаты Голда (Gold) показывают, что в три раза больше прогресса в интерактивной отладке достигается при первом взаимодействии каждого сеанса, чем при последующих взаимодействиях.⁸ Это убедительно свидетельствует о том, что мы не реализуем потенциал взаимодействия из-за отсутствия планирования сессий. Пришло время стряхнуть пыль со старой методики отладки.

Считаю, что надлежащее использование хорошей терминальной системы требует двух часов за письменным столом для каждого двухчасового сеанса за терминалом. Половина этого времени уходит на уборку после последнего сеанса: обновление журнала отладки, заполнение обновленных списков программ в системной записной книжке, объяснение странных явлений. Другая половина уходит на подготовку: планирование изменений, улучшений и разработку детальных тестов для следующего раза. Без такого планирования трудно оставаться продуктивным в течение двух часов. Без постсессионной уборки трудно поддерживать порядок последующих терминальных сессий систематическим и поступательным.

ТЕСТОВЫЕ СЛУЧАИ. Что касается разработки фактических отладочных процедур и тестовых случаев, то Груенбергер (Gruenberger) имеет

особенно хорошую трактовку⁹, и есть более короткие трактовки в других стандартных текстах.^{10,11}

ОТЛАДКА СИСТЕМЫ

Неожиданно трудной частью разработки системы программирования является системный тест. Я уже говорил о некоторых причинах, вызывающих как трудности, так и неожиданности в этом процессе. Исходя из всего этого можно быть уверенным в двух вещах: отладка системы займет больше времени, чем можно ожидать, и ее трудность оправдывает тщательно систематизированный и плановый подход. Давайте теперь посмотрим, что включает в себя такой подход.¹²

ЗАДЕЙСТВОВАТЬ ОТЛАЖЕННЫЕ КОМПОНЕНТЫ. Здравый смысл, если не обычная практика, говорит нам, что отладку системы следует начинать только после того, как ее части, по-видимому, работают.

Обычная практика отходит от этого двумя путями. Во-первых, это подход «прикрутить все вместе и попробовать». Указанный подход, похоже, основан на том, что в дополнение к дефектам компонентов будут существовать системные (то есть интерфейсные) ошибки. Чем скорее вы сложите части вместе, тем скорее проявятся системные ошибки. Несколько менее сложным является представление о том, что, используя части для тестирования друг друга, можно избежать большого количества тестовых строительных лесов. Оба этих утверждения являются очевидными, но опыт показывает, что они не являются полной правдой — использование чистых, отлаженных компонентов экономит гораздо больше времени при тестировании системы, чем возведение строительных лесов и тщательное тестирование компонентов.

Немного более тонким является подход *документированной ошибки*. Он говорит о том, что компонент готов войти в тест системы,

когда все недостатки будут *найденны*, но задолго до того времени, когда все будут *исправлены*. Затем в системном тестировании, так утверждает теория, будут известны ожидаемые влияния этих ошибок и указанные влияния можно игнорировать, концентрируясь на новых явлениях.

Все это является простым благонамеренным мышлением, изобретенным ради рационализации боли от сдвигов сроков. Никто *не* знает всех ожидаемых влияний известных ошибок. Если бы все было так просто, то системное тестирование не было бы сложным. Более того, исправление документированных компонентных ошибок, несомненно, приведет к появлению неизвестных ошибок, и в итоге системный тест будет запутан.

ВОЗВОДИТЬ СТРОИТЕЛЬНЫЕ ЛЕСА. Под строительными лесами я подразумеваю все программы и данные, построенные для отладочных целей, но никогда не использующиеся в конечном продукте. Утверждение, что строительные леса имеют объем вплоть до половины кода отлаживаемого продукта, недалеко от истины.

Одна из форм строительных лесов — фиктивный компонент, который состоит только из интерфейсов и, возможно, нескольких фальшивых данных или нескольких небольших тестовых случаев. Например, система может содержать программу сортировки, которая еще не завершена. Ее соседи могут быть протестированы с помощью фиктивной программы, которая просто читает и проверяет формат входных данных и извергает набор хорошо отформатированных бессмысленных, но упорядоченных данных.

Еще одна форма — *миниатюрный файл*. Очень распространенной формой системной ошибки является путаница форматов ленточных и дисковых файлов. Поэтому стоит создать несколько небольших файлов, которые имеют лишь несколько типичных записей, но содержат все дескрипторы, указатели и т. д.

Предельным случаем миниатюрного файла является *фиктивный файл*, которого на самом деле нет вообще. Язык управления заданиями OS/360 предоставляет такое средство обеспечения, и оно чрезвычайно полезно для отладки компонентов.

Еще одной формой строительных лесов являются вспомогательные программы. Генераторы тестовых данных, специальные аналитические распечатки, анализаторы таблиц перекрестных ссылок — все это примеры приспособлений и оснасток специального назначения, потребность в разработке которых может возникнуть.¹³

КОНТРОЛИРОВАТЬ ИЗМЕНЕНИЯ. Жесткий контроль во время тестирования является одним из впечатляющих методов отладки аппаратного обеспечения, и это также относится к программным системам.

Прежде всего, кто-то должен быть главным. Он и только он должен давать добро на изменение компонентов или замену одной версии на другую.

Тогда, как обсуждалось выше, должны быть контролируемые копии системы: одна зафиксированная копия последних версий, используемая для тестирования компонентов; одна тестируемая копия с установленными исправлениями; и рабочие копии, с которыми каждый может работать над своим компонентом, делая как исправления, так и расширения.

В моделях инженерии системы System/360 можно было встретить нерегулярные нити фиолетового провода среди обычных желтых проводов. Когда выявлялась ошибка, то делались две вещи. В системе разрабатывалось и устанавливалось быстрое исправление, поэтому тестирование могло продолжаться. Это изменение облекалось в фиолетовый провод, так, чтобы оно выделялось на общем фоне. Этот факт вносился в журнал. Тем временем готовился официальный документ о внесении исправлений, который запускался в жернова автоматизированного проектирования. В итоге

это выливалось в измененные чертежи и списки проводов и новую заднюю панель, в которой изменения были сделаны на печатной плате или желтыми проводами. Теперь физическая модель и бумага снова становились согласованными, а фиолетовый провод исчезал.

Программирование нуждается в методике фиолетового провода, и оно очень нуждается в жестком контроле и глубоком уважении к документу, который в конечном счете является продуктом. Жизненно важными составляющими такой методики являются протоколирование всех изменений в журнале и различие, проводимое очевидным образом в исходном коде, между заплатками на скорую руку и продуманными и документированными исправлениями.

ДОБАВЛЯТЬ ПО ОДНОМУ КОМПОНЕНТУ ЗА РАЗ. Эта заповедь тоже очевидна, но оптимизм и лень вынуждают нас ее нарушать. Чтобы следовать ему, требуются фиктивные программы и разное окружение, а это отнимает время. И в конце концов, не окажется ли, что вся эта работа не понадобится? Может быть, нет никаких ошибок?

Нет! Не поддавайтесь этому искушению! Именно в этом состоит систематичность системного тестирования. Нужно исходить из того, что ошибок будет много, и планировать упорядоченную процедуру их вылавливания.

Обратите внимание, что необходимо иметь полные тестовые случаи, тестируя частичные системы после добавления каждой новой части. И старые тесты, успешно выполненные на предыдущей частичной системе, должны быть выполнены повторно на новой, с целью ее регрессии.

КВАНТОВАТЬ ОБНОВЛЕНИЯ. По мере развития системы разработчики ее компонент время от времени будут приходить с новыми, горячими, версиями ее частей — более быстрыми, меньшего размера, более полными и предположительно менее дефектными. Замена рабочего компонента на новую версию требует такой же систематической

процедуры тестирования, как и добавление нового компонента, хотя это должно потребовать меньше времени, поскольку обычно доступны более полные и эффективные тестовые случаи.

Каждая команда, разрабатывающая еще один компонент, использует последнюю протестированную версию интегрированной системы в качестве тестового стенда для отладки своей части. Прделанная работа будет отброшена назад, если эта среда изменится. Конечно, она должна измениться. Но внесение изменений нужно производить квантами. Тогда у каждого пользователя будут промежутки продуктивной стабильности, прерываемые пакетным обновлением среды тестирования. Это оказывается значительно менее разрушительным, чем постоянные волнения и дрожь.

Леман и Беладди предлагают доказательства того, что кванты должны быть очень большими и редкими, или же очень маленькими и частыми.¹⁴ Последняя стратегия из перечисленных, согласно их модели, подвержена нестабильности в большей степени. Мой опыт это подтверждает: я никогда бы не стал рисковать, следуя этой стратегии на практике.

Квантованные изменения изящно вбирают в себя технологию фиолетового провода. Быстрое исправление сохраняется до следующего регулярного релиза компонента, который должен включать исправление в протестированной и документированной форме.



Антонио Канова. «Геракл убивает Лихаса», 1802. Лихас, вестник Геракла, убитый им за отравленный хитон. Источник: Википедия

14

И ГРЯНУЛ ГРОМ

Никто не любит гонца, приносящего
дурные вести.

Софокл

Каким образом проект может сме-
ститься на год?

...По одному дню за раз.

Когда кто-то слышит о катастрофическом отставании от графика на проекте, то представляет, что его постигла серия крупных бедствий. Между тем катастрофа обычно происходит из-за термитов, а не из-за торнадо; отставание происходило незаметно, но неумолимо. И действительно, с крупными бедствиями легче справиться; на них приходится реагировать большой силой, радикальной реорганизацией, изобретением новых подходов. По этому случаю поднимается вся команда.

Но каждодневное отставание труднее распознать, труднее предотвратить, труднее восполнить. Вчера ключевой человек заболел, и встреча не состоялась. Сегодня все машины вышли из строя, потому что молния ударила в силовой трансформатор здания. Завтра дисковые процедуры не начнут тестирование, потому что прибытие первого диска с завода опаздывает на неделю. Снегопад, работа в суде присяжных, семейные проблемы, экстренные встречи с клиентами, проверки начальства — список можно продолжать и продолжать. Каждое из этих событий задерживает работу лишь на полдня или на день. И растет отставание от графика, каждый раз еще на один день.

ВЕХИ ИЛИ ЖЕРНОВА?

Как контролировать крупный проект по жесткому графику? Первый шаг — *иметь* график. Каждое событие из списка событий, именуемых контрольными точками, имеет дату. Подбор дат является задачей оценивания, которая уже обсуждалась и в решающей степени зависит от опыта.

Для подбора контрольных точек есть только одно релевантное правило. Контрольные точки должны быть конкретными, специфичными, измеримыми событиями, точно определенными. Вот несколько примеров, как делать не надо. Написание кода «на

90% завершено» в течение половины общего времени написания кода. Отладка «на 99% завершена» почти всегда. «Планирование завершено» можно объявить почти произвольно.¹

С другой стороны, конкретные контрольные точки — это 100-процентные события. «Спецификации подписаны архитекторами и имплементаторами», «исходный код написан и на 100% завершен, отперфорирован, введен в дисковую библиотеку», «отлаженная версия проходит все тестовые случаи». Такие конкретные вехи разграничивают расплывчатые этапы планирования, кодирования и отладки.

Гораздо важнее, чтобы контрольные точки были четкими и недвусмысленными, а не просто легко поддавались проверке. Человек редко будет лгать о прогрессе контрольной точки, *если* контрольная точка настолько четкая, что он не сможет себя обмануть. А вот если веха расплывчата, начальник часто воспринимает доклад иначе, чем тот, кто ему докладывает. Дополняя Софокла, скажем, что никто не любит и сам приносить дурные вести, поэтому они смягчаются без злого намерения ввести в заблуждение.

Два интересных исследования поведения, связанного с оцениванием у правительственных подрядчиков на крупномасштабных строительных проектах, показывают, что:

1. Оценки продолжительности деятельности, тщательно составляемые и пересматриваемые каждые две недели до ее начала, не претерпевают существенных изменений по мере приближения времени начала, независимо от того, насколько ошибочными они в конечном счете оказываются.
2. *Во время* деятельности завышенные оценки продолжительности неуклонно снижаются по мере продолжения деятельности.
3. В процессе деятельности *заниженные оценки* существенно не изменяются вплоть до примерно трех недель до запланированной даты завершения соответствующих им работ.²

Четкие контрольные точки являются подарком для команды — она понимает, чего следует ожидать от менеджера. Нечеткая контрольная точка является тяжким бременем, с которым нужно жить. Это жернов, который перемалывает мораль, ибо вводит человека в заблуждение о потерянном времени до тех пор, пока уже невозможно все будет исправить. А хроническое отставание от графика является убийцей морали.

«ДРУГАЯ ЧАСТЬ РАБОТЫ ВСЕ РАВНО ЗАПАЗДЫВАЕТ»

График отстает на день, ну и что? Кого волнует однодневное отставание? Нагоним позже. Другая часть работы, в которую вписывается наша, все равно запаздывает.

Бейсбольный менеджер признает нефизический талант, *энергию*, как неотъемлемый дар великих игроков и великих команд. Это свойство бегать быстрее, чем нужно, двигаться быстрее, чем нужно, стараться больше, чем нужно. Энергия важна и для выдающихся команд программистов. Она обеспечивает упругость, резервную мощность, позволяющие команде справиться с повседневными неприятностями, предвосхищать мелкие беды и уберечься от них. Рассчитанный отклик, измеренное усилие охлаждают энергию. Как мы уже видели, человек просто *должен* волноваться от однодневного отставания графика, ибо они — элементы катастрофы.

Но не все однодневные отставания одинаково катастрофичны. Поэтому необходим некоторый расчет отклика, чтобы остудить пыл. Как определить, какие смещения графика имеют значение? Ничто не может заменить диаграмму PERT или метод критического пути. Этот метод показывает, кто чего ждет. Он показывает, кто находится на критическом пути, где любое отставание сдвигает конечную дату. Он также показывает, насколько работа может отстать от графика, прежде чем перейдет на критический путь.

Метод PERT, строго говоря, представляет собой развитие метода планирования критического пути, где каждое событие оценивается трижды с точки зрения времени, соответствующего различным вероятностям достижения предполагаемых дат. Я не считаю, что это уточнение стоит дополнительно пояснять, но для краткости я буду называть любую сеть критического пути диаграммой PERT.

Подготовка диаграммы PERT является наиболее ценной частью ее использования. Сборка сети, определение зависимостей и оценивание этапов эстафеты — все это требует очень специфического планирования на самой ранней стадии проекта. Первая диаграмма всегда является ужасной, и человек изобретает и изобретает, создавая вторую.

По мере того как проект продолжается, диаграмма PERT дает ответ на деморализующее оправдание: «другая часть работы все равно запаздывает». Она показывает, насколько необходима энергия, чтобы удерживать свою собственную часть работы в стороне от критического пути, и подсказывает способы наверстать упущенное время в другой части работы.

ПОД КОВЕР

Когда менеджер низшего звена видит, что его маленькая команда отстает, он обычно не бежит к боссу со своим горем. Возможно, команда сумеет наверстать время, либо он сможет что-нибудь придумать или реорганизовать для решения проблемы. Тогда зачем беспокоить этим босса? Пока все идет хорошо. Решение таких проблем — это именно то, для чего существует менеджер низшего звена. И к тому же у босса достаточно реальных забот, требующих его действий, чтобы он не искал другие. Так что вся грязь заматается под ковер.

Но каждому начальнику нужны два вида информации: исключения из плана, требующие действий, и картина о состоянии дел для осведомленности.³ Для этой цели ему нужно знать состояние дел всех своих команд. Получить истинную картину об этом состоянии очень трудно.

Интересы менеджера низшего звена и интересы босса вступают в противоречие. Менеджер низшего звена опасается, что если он сообщит о своей проблеме, то босс будет действовать в соответствии с ней. Тогда его действия вытеснят функции менеджера, преуменьшат его полномочия, испортят другие его планы. Поэтому до тех пор, пока менеджер думает, что он может решить проблему в одиночку, он не говорит боссу.

У босса в распоряжении есть два способа узнать, что под ковром. Оба должны быть использованы. Первый — уменьшить конфликт ролей и стимулировать открытие информации. Другой — сдернуть ковер.

СНИЖЕНИЕ КОНФЛИКТА РОЛЕЙ. Прежде всего босс должен проводить различие между информацией, требующей действия, и информацией о состоянии дел. Он должен настроить себя *не* предпринимать действий по проблемам, которые могут быть решены его менеджерами, и *никогда* не предпринимать действий еще только лишь изучая положение дел. Я знал одного начальника, который неизменно снимал трубку и начинал давать указания, не дочитав до конца первый абзац отчета о состоянии дел. При таких действиях вам обеспечено утаивание полных данных.

И наоборот, когда менеджер знает, что его босс примет отчет о состоянии дел без паники или вмешательства, то будет давать честный отчет.

Процесс идет успешно, если начальник подчеркивает, что совещания, отчеты и конференции носят характер изучения состояния дел,

а не принятия мер по проблемам, и ведет себя соответствующим образом. Очевидно, что можно созвать *совещание по принятию мер по решению проблемы* как следствие *совещания по состоянию дел*, если он считает, что проблема вышла из-под контроля. Но по крайней мере все знают, что происходит, и босс дважды подумает, прежде чем взять все в свои руки.

СДЕРГИВАНИЕ КОВРА. Тем не менее необходимо иметь методику пересмотра, с помощью которой истинное состояние становится известным независимо от наличия стремления к сотрудничеству. Основой для такого обзора является диаграмма PERT с ее частыми четкими контрольными точками. В крупном проекте можно потребовать еженедельного изучения какой-либо части ее, рассматривая всю диаграмму раз в месяц или около того.

Отчет, показывающий контрольные точки и фактические завершения работ, является ключевым документом. На рис. 14.1 показан фрагмент такого отчета. Этот отчет вскрывает некоторые проблемы. Утверждение спецификаций просрочено по нескольким компонентам. Утверждение справочного руководства (SLR) просрочено на еще одном, и один из них запаздывает с выходом из первого состояния (Alpha) независимо от проведенного теста продукта. Таким образом, такой доклад служит повесткой дня для совещания 1 февраля. Все знают вопросы, и менеджер по разработке компонентов должен быть готов объяснить, в чем причина отставания, когда работа будет закончена, какие шаги он предпринимает и какая помощь требуется, если она вдруг понадобится от начальника или других команд.

Высоцкий из *Bell Telephone Laboratories* добавляет следующее наблюдение:

Я нашел удобным переносить и «запланированные по графику», и «оценочные» даты в отчет о контрольной точке. Запланированные даты принадлежат менед-

жеру проекта и представляют собой согласованный план работы для проекта в целом, который априори является разумным планом. Оценочные даты принадлежат менеджеру низшего звена, который компетентен в рассматриваемой части работы и дает свое лучшее суждение о том, когда это действительно произойдет, учитывая имеющиеся у него ресурсы, и когда он получил (или взял на себя обязательства по доставке) необходимые результаты по работам, от которых он зависит. Менеджер проекта должен осторожно относиться к оцениваемым датам и стремиться к получению точных, неискаженных оценок, а не утешительно-оптимистичных или перестраховочно-консервативных данных. Если эта позиция утвердится в умах, то менеджер проекта действительно сможет предвидеть, что он попадет в беду, если не предпримет каких-нибудь мер.⁴

Подготовка диаграммы PERT является задачей руководителя и подчиняющихся ему менеджеров. Ее обновление, пересмотр и отчетность по ней требуют внимания небольшой (от одного до трех человек) группы сотрудников, являющейся представителем руководителя. Такая команда «планирования и осуществления контроля» имеет неоценимое значение для крупного проекта. У нее нет никаких полномочий, кроме как опрашивать всех линейных менеджеров об установке или изменении контрольных точек и их выполнении. Поскольку группа «планирования и осуществления контроля» занимается всей бюрократической работой, задачей линейных менеджеров остается принятие решений.

У нас была опытная, полная энтузиазма и дипломатичная группа по «планированию и осуществлению контроля», возглавляемая А. М. Пьетрасантой (A. M. Pietrasanta), проявившим значительные изобретательные способности для разработки эффективных, но ненавязчивых методов контроля. В результате я обнаружил, что его группа пользуется всеобщим уважением и хорошим отношением. Это немалое достижение для группы, которая по природе своей должна вызывать раздражение.

SYSTEM/360 SUMMARY STATUS REPORT 05/360 LANGUAGE PROCESSORS & SERVICE PROGRAMS AS OF FEBRUARY 01, 1965									
PROJECT	LOCATION	COMMITMENT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPL AVAILABLE APPROVED	ALPHA TEST EXIT COMPLETE	COMP TEST COMPLETE	SYS TEST COMPLETE	4-REVISED PLANNED DATE	
								BULLETIN APPROVED	BETA TEST EXIT
OPERATING SYSTEM									
12K DESIGN LEVEL (E)									
ASSEMBLY	SAN JOSE	04/--/A C 12/31/5	10/13/A C 01/11/6	11/13/A C 11/18/A A	01/15/5 C 02/22/5				09/01/5 11/30/5
FORTRAN	POK	04/--/A C 12/31/6	10/28/A C 01/22/5	12/17/A C 12/19/A A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	ENDICOTT	04/--/A C 12/31/5	10/28/A C 01/20/5 A	11/11/A C 12/08/A A	01/15/5 C 02/22/5				09/01/5 11/30/5
RPG	SAN JOSE	04/--/A C 12/31/5	10/28/A C 01/05/5 A	12/02/A C 01/16/5 A	01/15/5 C 02/22/5				09/01/5 11/30/5
UTILITIES	TIME/LIFE	04/--/A C 12/31/5	06/24/A C	11/30/A A					09/01/5 11/30/5
SORT 1	POK	04/--/A C 12/31/5	10/28/A C 01/11/5	11/12/A C 11/30/A A	01/15/5 C 03/22/5				09/01/5 11/30/5
SORT 2	POK	04/--/A C 06/30/6	10/28/A C 01/11/5	11/12/A C 11/30/A A	01/15/5 C 03/22/5				03/01/6 05/30/6
44K DESIGN LEVEL (F)									
ASSEMBLY	SAN JOSE	04/--/A C 12/31/5	10/28/A C 01/11/5	11/13/A C 11/18/A A	02/15/5 03/22/5				09/01/5 11/30/5
COBOL	TIME/LIFE	04/--/A C 06/30/6	10/28/A C 01/20/5 A	11/17/A C 12/08/A A	02/15/5 03/22/5				03/01/6 05/30/6
MPL	MURSLEY	04/--/A C 03/31/6	10/28/A C						
2250	KINGSTON	03/30/4 C 03/31/6	11/05/A C 12/08/A C	01/12/5 C 01/29/5	01/04/5 C 01/29/5				01/03/6 ME
2280	KINGSTON	06/30/4 C 09/30/6	11/05/A C	01/29/5 04/30/5	04/01/5 04/30/5				01/28/6 ME
200K DESIGN LEVEL (H)									
ASSEMBLY	TIME/LIFE	10/28/A C							
FORTRAN	POK	04/--/A C 06/30/6	10/28/A C 01/11/5	11/11/A C 12/10/A A	02/15/5 03/22/5				03/01/6 05/30/6
MPL	MURSLEY	04/--/A C 03/31/7	10/28/A C		07/--/5				01/--/7 10/15/5
MPL H	POK	04/--/A C 03/30/A C			02/01/5 04/01/5				12/15/5

Рис. 14.1

Вложение скромного количества квалифицированных усилий в функции планирования и осуществления контроля приносит весьма большую отдачу. Это имеет гораздо большее значение в выполнении проекта, чем если бы эти люди работали непосредственно над разработкой программ продукта, так как группы «планирования и осуществления контроля» представляют собой сторожевого пса, который делает незаметные задержки видимыми и который указывает на критически важные положения. Это система раннего обнаружения потери года, происходящей день за днем.



Реконструкция Стоунхенджа, крупнейшего в мире недокументированного компьютера.
Источник: Википедия

15

ДРУГАЯ СТОРОНА

Мы не владем тем, чего не понимаем.

Гёте

О, дайте мне выступить комментатором, скользящим по поверхности и будоражащим умы.

Краббе

Компьютерная программа — это послание человека машине. Строго упорядоченный синтаксис и скрупулезные определения — все это существует для того, чтобы сделать намерение ясным для бездумной машины.

Но у написанной программы есть и другая сторона, та, которая рассказывает свою историю пользователю-человеку. Это требуется даже для частных, персональных программ; память подведет автора-пользователя, и ему потребуется освежить детали своей работы.

Насколько же еще более насущной является документация публичной программы, пользователь которой находится на расстоянии от автора как во времени, так и в пространстве! Другая сторона программного продукта, обращенная к потребителю, имеет абсолютно одинаковую важность, как и сторона, обращенная к машине.

Большинство из нас по-тихому устраивали разнос далекому и анонимному автору какой-нибудь плохо документированной программы. И поэтому многие из нас пытались привить новым программистам отношение к документации, которое вдохновляло бы на всю жизнь преодолевать лень и давление графика. По большому счету, мы потерпели неудачу. Думаю, мы использовали не те методы.

Томас Дж. Уотсон-старший (Thomas J. Watson, Sr.) рассказывает следующее о своем первом опыте работы продавцом кассовых аппаратов в северной части штата Нью-Йорк. Преисполненный энтузиазма, он выехал на своем фургоне, нагруженном кассовыми аппаратами. Он старательно объездил свою территорию, но ни одного так и не продал. Удрученный, он доложил о результатах боссу. Менеджер по продажам послушал, а затем сказал: «Помоги мне погрузить несколько аппаратов в фургон, заводи мотор и поехали». Они так и сделали: оба обзванивали одного клиента за другим, и старый босс *показывал, как* продавать кассовые аппараты. Урок был усвоен.

В течение нескольких лет я читал лекции по инженерии программного обеспечения о необходимости и уместности хорошей документации, убеждая студентов все горячее и красноречивее. И это не работало. Я предположил, что они поняли, как правильно составлять документацию, но не делали этого по недостатку рвения. Тогда я попытался погрузить в фургон несколько кассовых аппаратов, то есть *показать* им, как делается работа. Это оказалось гораздо успешнее. Поэтому остальная часть этого эссе будет меньше увещевать и концентрироваться на вопросе «как» хорошей документации.

КАКАЯ ДОКУМЕНТАЦИЯ ТРЕБУЕТСЯ?

Для случайного пользователя программы, для пользователя, который должен зависеть от программы, и для пользователя, который должен адаптировать программу под изменения обстоятельств или цели, требуются разные уровни документации.

ЧТОБЫ ПОЛЬЗОВАТЬСЯ ПРОГРАММОЙ. Каждому пользователю необходимо описание программы на естественном языке. По большей части документация страдает отсутствием общего обзора. Деревья описаны, кора и листья прокомментированы, но карты леса нет. Чтобы написать полезное описание в текстовой форме, следует отойти назад, а затем медленно приближаться:

1. *Предназначение.* Какова главная функция, которая является причиной появления этой программы?
2. *Среда.* На каких машинах, конфигурациях аппаратного обеспечения и конфигурациях операционных систем она будет работать?
3. *Область и диапазон.* Какая область входных данных допустима? Какой диапазон выходных данных может законно появиться?

4. *Реализованные функции и используемые алгоритмы.* Что именно она делает?
5. *Форматы ввода-вывода,* точные и полные.
6. *Операционные инструкции,* включая поведение при нормальном и ненормальном завершении работы, в том виде, в каком результат отражается на консоли и на других выходах.
7. *Опции.* Какие варианты выбора есть у пользователей в плане функций? Как именно определяются эти варианты?
8. *Время выполнения.* Сколько времени требуется на выполнение задачи указанного размера в указанной конфигурации?
9. *Точность и проверка.* Насколько точными должны быть ответы? Какие средства проверки точности встроены?

Часто вся эта информация может быть изложена на трех-четырех страницах. Она требует пристального внимания к краткости и точности. Большая часть этого документа должна быть подготовлена до того, как будет написана программа, поскольку документ воплощает основные планируемые решения.

ЧТОБЫ ВЕРИТЬ ПРОГРАММЕ. Описание того, как она используется, должно быть дополнено некоторым описанием того, как можно узнать, что она работает. Это подразумевает тестовые случаи.

Каждая копия поставляемой программы должна включать в себя несколько небольших тестовых случаев, которые можно регулярно использовать, чтобы убедить пользователя в том, что он работает с правильной копией, аккуратно загруженной в машину.

Затем требуются более тщательные тестовые случаи, которые обычно выполняются только после того, как программа была модифицирована. Они распадаются на три части области входных данных:

1. Основные случаи, которые проверяют главные функции программы на обычно встречающихся данных.
2. Примеры на грани допустимого, проверяющие границы области входных данных и убеждающие, что работают наибольшие значения, наименьшие значения и все допустимые исключения.
3. Примеры за границей допустимого, проверяющие границы с обратной стороны и убеждающие, что недопустимые значения вызывают правильные диагностические сообщения.

ЧТОБЫ МОДИФИЦИРОВАТЬ ПРОГРАММУ. Адаптация программы или ее исправление требуют значительно большего объема информации. Конечно, требуется полная детализация, которая содержится в хорошо прокомментированном листинге. Для того, кто собирается модифицировать программу, а также кто редко ее использует, остро необходимо ясное, четкое общее описание, на этот раз внутренней структуры. Каковы составляющие такого общего описания?

1. Блок-схема или граф структуры подпрограммы. Подробнее об этом позже.
2. Полные описания используемых алгоритмов либо ссылки на такие описания в литературе.
3. Объяснение общей схемы всех используемых файлов.
4. Обзор организации прохождения данных — последовательности, в которой данные или программы загружаются с ленты или диска и описание того, что делается на каждом ходе.
5. Обсуждение модификаций, предусмотренных в оригинальном дизайне, характер и расположение перехватчиков (хуков) и выходов, а также дискурсивное обсуждение идей оригинального автора о том, какие модификации могут быть желательными и как можно было бы поступить. Также полезны его наблюдения о скрытых ловушках.

ПРОКЛЯТИЕ БЛОК-СХЕМЫ

Блок-схема чаще всего является лишней частью программной документации. Многие программы вообще не нуждаются в блок-схемах, и немногие программы нуждаются в чем-то большем, чем одностраничная блок-схема.

Блок-схемы показывают структуру принятия решений в программе, являясь только одним аспектом ее структуры. Они показывают структуру принятия решений довольно элегантно, когда блок-схема находится на одной странице, но общее видение совершенно разрушается, когда у вас несколько страниц, сшитых вместе с пронумерованными выходами и соединителями.

Одностраничная блок-схема солидной программы становится по существу схемой структуры программы, а также фаз или шагов. Как таковая она очень удобна. На рис. 15.1 показан такой структурный граф подпрограммы.

Конечно, такой структурный граф не подчиняется мучительно выработанным стандартам ANSI по организации блок-схем и в них не нуждается. Все правила о формах блоков, соединителях, нумерации и т. д. нужны только для того, чтобы придать разборчивость подробным блок-схемам.

Однако детальная и методичная блок-схема является устаревшим неудобством, пригодным только для того, чтобы обучать новичков алгоритмическому мышлению. Введенные Голдштайном и фон Нейманом прямоугольники вместе со своим содержимым служили языком высокого уровня, объединяя непостижимые операторы машинного языка в осмысленные кластеры. Как с самого начала признал Айверсон², в систематическом языке высокого уровня кластеризация уже выполнена и каждый блок содержит инструкцию языка (рис. 15.2). Тогда сами блоки становятся не более чем утомительным и расходующим пространство упражнением в чер-

чении; с таким же успехом их можно убрать. И тогда не остается ничего, кроме стрелок. Стрелки, соединяющие инструкцию с ее преемницей, являются избыточными; удалим их. Останутся только переходы GOTO. И если следовать хорошей практике и использовать блочную структуру языка с целью минимизации переходов GOTO, то стрелок будет не так много, но они будут очень помогать пониманию. С таким же успехом их можно нарисовать на листинге и полностью исключить блок-схему.

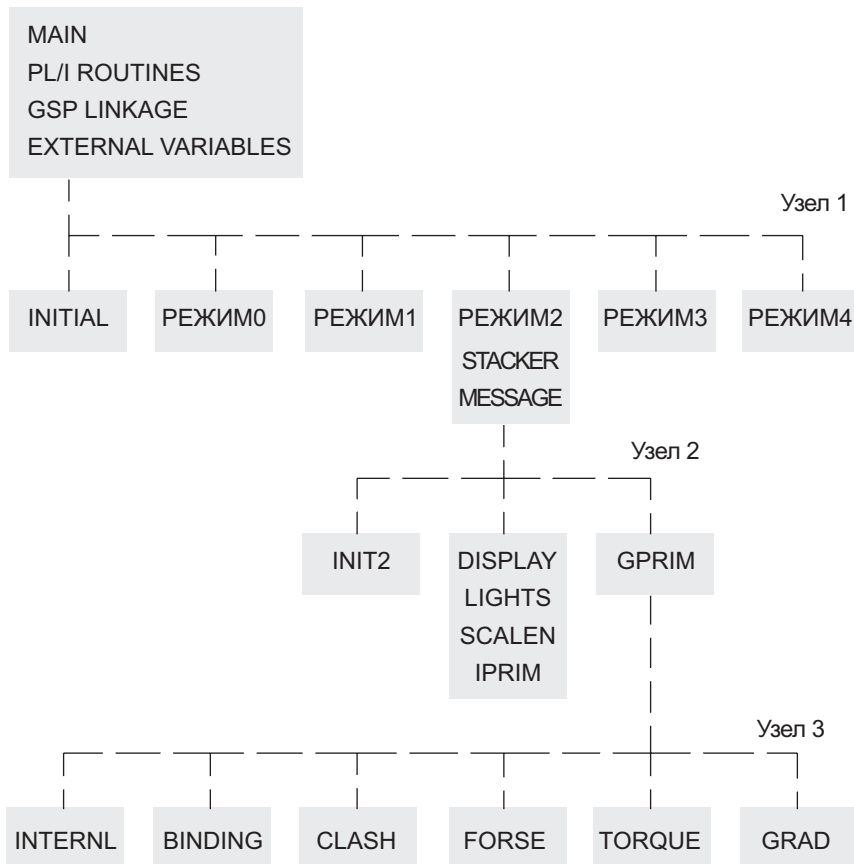
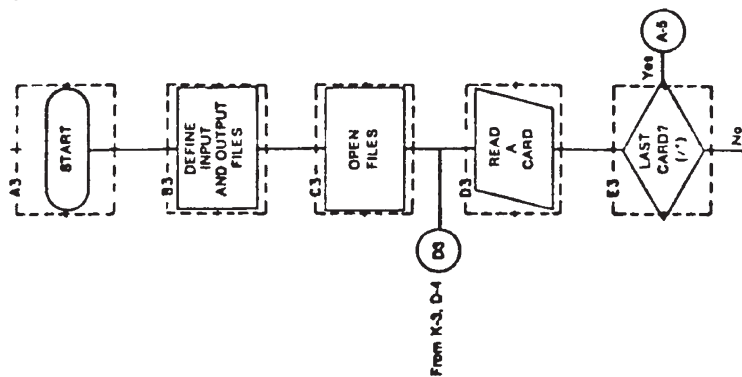


Рис. 15.1. Граф структуры программы (с разрешения У. В. Райта)



PGM1: PROCEDURE OPTIONS (MAIN);

```

DECLARE SALEFL FILE
RECORD
  INPUT
  ENVIRONMENT (F180) MEDIUM (SYSIPT, 2501));
DECLARE PRINT4 FILE
RECORD
  OUTPUT
  ENVIRONMENT (F132) MEDIUM (SYSLS1,1403) C7LASA);
DECLARE 01 SALESCARD,
  03 BLANK1,
  03 SALESNUM
  03 NAME
  03 BLANK2
  03 CURRENT_SALES
  03 BLANK3
  03 SALESLIST,
  03 CONTROL
  03 SALESNUM_OUT
  03 FILLER1
  03 NAME_OUT
  03 FILLER2
  03 CURRENT_OUT
  03 FILLER3
  03 PERCENT
  03 SIGN
  03 FILLER4
  03 COMMISSION
  03 FILLERS
  CHARACTER (91),
  PICTURE '9999',
  CHARACTER (251),
  CHARACTER (71),
  PICTURE '9999999',
  CHARACTER (291),
  CHARACTER (11) INITIAL (' '),
  PICTURE 'Z229',
  CHARACTER (51) INITIAL (' '),
  CHARACTER (251),
  CHARACTER (51) INITIAL (' '),
  PICTURE 'Z,222V,99',
  CHARACTER (51) INITIAL (' '),
  PICTURE '29',
  CHARACTER (11) INITIAL ('9'),
  CHARACTER (51) INITIAL (' '),
  PICTURE 'Z,222V,99',
  CHARACTER (63) INITIAL (' ');
  
```

OPEN FILE (SALEFL),FILE (PRINT4);

ON ENDFILE (SALEFL) GO TO ENDOFJOB1

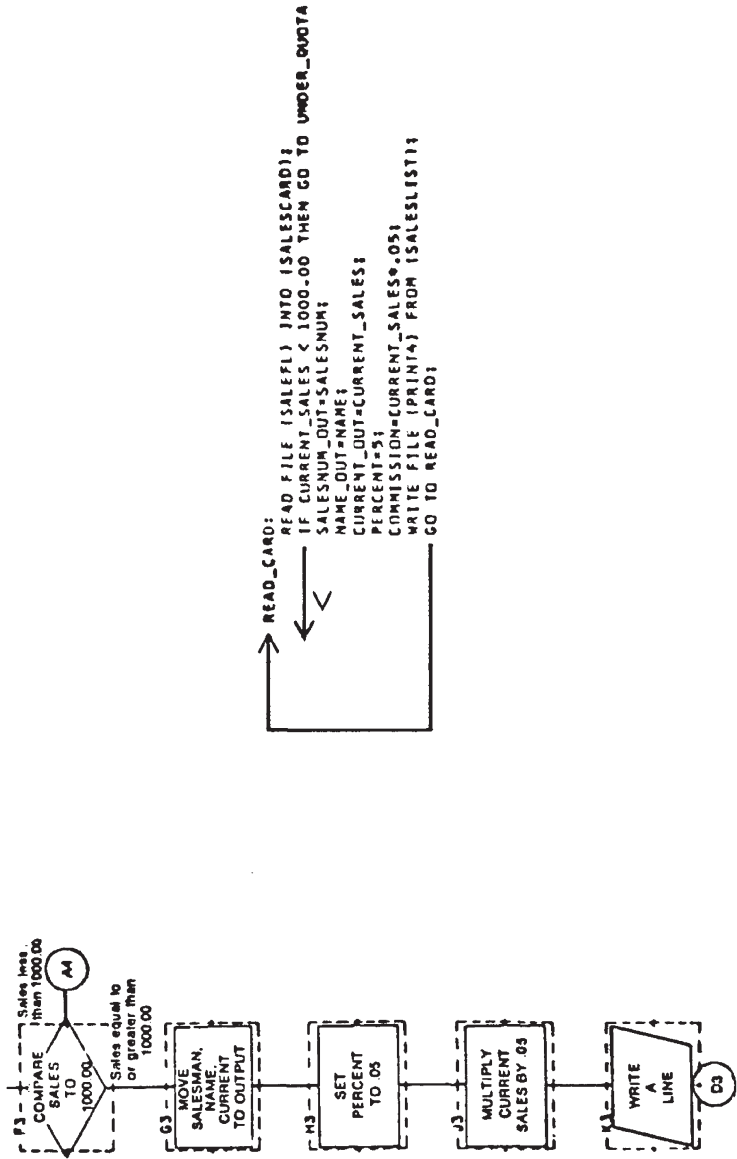


Рис. 15.2. Сравнение блок-схемы и соответствующей программы на PL/1 (сокращено и адаптировано)

На самом деле о блок-схемах больше говорят, чем пользуются. Я никогда не видел опытного программиста, который составлял бы подробные блок-схемы, прежде чем начинать писать программу. Там, где требуют стандарты организации, блок-схемы почти всегда делаются постфактум. Многие центры с гордостью используют машинные программы для генерирования этого «незаменимого инструмента разработки» из готового кода. Я думаю, что этот универсальный опыт не является постыдным и прискорбным отступлением от хорошей практики, признаваться в котором можно лишь с нервным смешком. Вместо этого он является применением здравого смысла и кое-чему учит нас о полезности блок-схем.

Апостол Петр сказал о новообращенных язычниках и иудейском законе: «Зачем возлагать на [их] спины груз, который ни наши предки, ни мы сами не смогли нести?» (Деяния 15: 10). Я бы сказал то же самое о программистах-новичках и устаревшей практике блок-схем.

САМОДОКУМЕНТИРОВАННЫЕ ПРОГРАММЫ

Базовый принцип обработки данных учит о безрассудстве попыток поддерживать синхронность независимых файлов. Гораздо лучше совместить их в один файл, где каждая запись содержит всю информацию, которую оба файла содержат относительно заданного ключа.

Однако наша практика в документации по программированию нарушает наше собственное учение. Мы, как правило, стараемся поддерживать машиночитаемую форму программы и независимый набор человекочитаемой документации, состоящей из текста и блок-схем.

Результаты этого подтверждают мысль о неразумности поддержки независимых файлов. Всем известно, что пояснения к коду доку-

ментируются плохо, а поддерживаются и того хуже. Изменения, вносимые в программу, не появляются на бумаге быстро, точно и неизменно.

Решение, я думаю, заключается в слиянии файлов: встраивании документации в исходный код программы. Это одновременно и мощный стимул к правильному сопровождению, и гарантия того, что документация всегда будет удобна пользователю программы. Такие программы называются *самодокументированными*.

Теперь ясно, что это является неудобным (но не невозможным), если нужно включить блок-схемы. Но учитывая устаревание блок-схем и доминирующее использование языка высокого уровня, становится разумным комбинировать программу и документацию.

Использование исходной программы в качестве среды документации накладывает некоторые ограничения. С другой стороны, непосредственный доступ читателя документации к каждой строке программы открывает возможность для новых технологий.

Пришло время разработать принципиально новые подходы и методы документирования кода.

В качестве первостепенного целевого критерия мы должны попытаться снизить бремя документации, бремя, которое ни мы, ни наши предшественники не смогли нести успешно.

подход. Первая идея — использовать части программы, которые так или иначе должны иметь силу языка программирования как носителя максимального объема документации. Соответственно, метки, операторы объявления и символические имена включают в задачу передать читателю как можно больше смысла.

Вторая идея — максимально использовать пространство и формат с целью улучшения читаемости и демонстрации подчиненности и вложенности.

Третья идея — вставить необходимую текстовую документацию в программу в виде абзацев с комментариями. Большинство программ, как правило, имеют достаточно построчных комментариев; эти программы, производимые в соответствии с жесткими организационными стандартами ради «хорошего документирования», часто имеют их слишком много. Однако в этих программах, как правило, мало абзачных комментариев, которые действительно придают понятность и общее видение.

Поскольку документация встраивается в используемые программой структуру, имена и форматы, значительную часть этой работы необходимо проделать, когда программу только начинают писать. И именно тогда ее *следует* написать. Поскольку подход к самодокументированию уменьшает лишнюю работу, становится меньше препятствий к тому, чтобы им воспользоваться.

НЕКОТОРЫЕ ТЕХНИКИ. На рис. 15.3 показана самодокументированная программа на PL/1.³ Числа в кружочках не являются ее частью; они представляют собой метадокументацию, индексы для ссылки из данного обсуждения.

1. Используйте отдельное имя задания для каждого запуска и ведите журнал, показывающий, что было выполнено, когда и результаты. Если имя состоит из мнемонической части (здесь *QLT*) и числового суффикса (здесь 4), то суффикс может использоваться в качестве номера запуска, связывая листинги и журнал вместе. Этот метод требует новой карточки задания для каждого запуска, но они могут быть изготовлены партиями, дублируя общую информацию.
2. Используйте имя программы, которое является мнемоническим, но также содержит идентификатор версии. Предполагайте, что будет несколько версий. Здесь индекс — это младшая цифра 1967 года.
3. Встройте текстовое описание в качестве комментариев к PROCEDURE.


```

① //QLT7 JOB ...
② QLTST7: PROCEDURE (V);

③ /*A SORT SUBROUTINE FOR 2500 6-BYTE FIELDS, PASSED AS THE VECTOR V. A
/*SEPARATELY COMPILED, NOT-HAIN PROCEDURE, WHICH MUST USE AUTOMATIC CORE
/*ALLOCATION.
/*
④ /*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA PROCESSING.
/*PROGRAM 7.23, P. 350. THAT ALGORITHM IS REVISED AS FOLLOWS:
⑤ /* STEPS 2-12 ARE SIMPLIFIED FOR N=2.
/* STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT VECTOR.
/* THE WHOLE FIELD IS USED AS THE SORT KEY.
/* MINUS INFINITY IS REPRESENTED BY ZEROS.
/* PLUS INFINITY IS REPRESENTED BY ONES.
/* THE STATEMENT NUMBERS IN PROG. 7.23 ARE REFLECTED IN THE STATEMENT
/* LABELS OF THIS PROGRAM.
/* AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION OF A FEW LINES.
/*
/*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE THE
/*INITIALIZATION OF T. IF THE SIZE EXCEEDS 4096, CHANGE THE SIZE OF T, TOO.
/*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V.
/*
/*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR.
/******

⑥ /* LEGEND (ZERO-ORIGIN INDEXING) */
DECLARE
(N, /*INDEX FOR INITIALIZING T */
I, /*INDEX OF ITEM TO BE REPLACED */
J, /*INITIAL INDEX OF BRANCHES FROM NODE I */
K) BINARY FIXED, /*INDEX IN OUTPUT VECTOR */

(MINF, /*MINUS INFINITY */
(PINF) BIT (48), /*PLUS INFINITY */

V (*) BIT (*), /*PASSED VECTOR TO BE SORTED AND RETURNED */

T (0:8190) BIT (48); /*WORKSPACE CONSISTING OF VECTOR TO BE SORTED, FILLED*/
/*OUT WITH INFINITIES, PRECEDED BY LOWER LEVELS */
/*FILLED UP WITH MINUS INFINITIES */

/* NOW INITIALIZATION TO FILL DUNNY LEVELS, TOP LEVEL, AND UNUSED PART OF TOP*/
/* LEVEL AS REQUIRED. */

⑦ INIT: MINF= (48) '0'B;
PINF= (48) '1'B;

DO L= 0 TO 4094; T(L) = MINF; END;
DO L= 0 TO 2499; T(L+4095) = V(L); END;
DO L=6595 TO 8190; T(L) = PINF; END;

⑧ K0: K = -1;
K1: I = 0;
K3: J = 2*I+1; /*SET J TO SCAN BRANCHES FROM NODE I.
K7: IF T(J) <= T(J+1) /*PICK SMALLER BRANCH
THEN
DO:
⑨ K11: T(I) = T(J); /*REPLACE
K13: IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT
/* IS FINISHED
K12: I = J; /*SET INDEX FOR HIGHER LEVEL
END;
ELSE
DO:
K11A: T(I) = T(J+1); /*
K13A: IF T(I) = PINF THEN GO TO K16; /*
K12A: I = J+1; /*
END;
K14: IF 2*I < 8191 THEN GO TO K3; /*GO BACK IF NOT ON TOP LEVEL
K15: T(I) = PINF; /*IF TOP LEVEL, FILL WITH INFINITY
K16: IF T(0) = PINF THEN RETURN; /*TEST END OF SORT
K17: IF T(0) = MINF THEN GO TO K1; /*FLUSH OUT INITIAL DUNNIES
K18: K = K+1; /*STEP STORAGE INDEX
T(K) = T(0); GO TO K1; ⑩ /*STORE OUTPUT ITEM
END QLTST7;

```

Рис. 15.3. Самодокументированная программа

4. Ссылайтесь на стандартную литературу, документируя базовые алгоритмы, где это возможно. Это экономит место, обычно указывает на гораздо более полную трактовку, чем можно было бы обеспечить, и позволяет знающему читателю пропустить ее с уверенностью, что он вас понимает.
5. Покажите взаимоотношение с книжным алгоритмом:
 - а) изменения; б) специализация; в) представление.
6. Объявите все переменные. Используйте мнемонические имена. Используйте комментарии, конвертируя объявление `DECLARE` в полноценную легенду. Обратите внимание, что оно уже содержит имена и структурные описания, его нужно только дополнить описаниями *цели*. Делая это здесь, можно избежать повторения имен и структурных описаний в отдельной трактовке.
7. Поставьте метку в начале инициализации.
8. Отметьте группы инструкций языка метками с целью показать соответствия с инструкциями в описании алгоритма в литературе.
9. Используйте отступы для отображения структуры и группирования.
10. Добавьте в листинг от руки стрелки логических переходов. Они очень полезны при отладке и изменении. Они могут быть встроены в правое поле пространства комментариев и сделаны частью машиночитаемого текста.
11. Вставьте строчные комментарии для пояснения всего, что неочевидно. При использовании изложенных выше приемов они окажутся короче и малочисленней, чем обычно.
12. Размещайте несколько инструкций языка на одной строке или одну инструкцию на нескольких строках в соответствии с группировкой по смыслу, чтобы показать соответствие с описанием другого алгоритма.

ВОЗРАЖЕНИЯ. Каковы недостатки такого подхода к документированию? Они существуют, и в прежние времена были существенными, однако сейчас становятся мнимыми.

Самым серьезным возражением является увеличение размера исходного кода, который должен храниться. По мере того как дисциплина все больше и больше продвигается к онлайн-хранению исходного кода, это становится все более важным соображением. Я нахожу, что более кратко комментирую программу на APL, которая будет расположена на диске, чем на PL/1, которую буду хранить на перфокартах.

Но одновременно мы движемся и к онлайн-хранению текстовых документов, обращаясь к ним и обновляя их с помощью редактирования текста посредством компьютера. Как показано выше, объединение текста и программы *сокращает* общее число символов, подлежащих хранению.

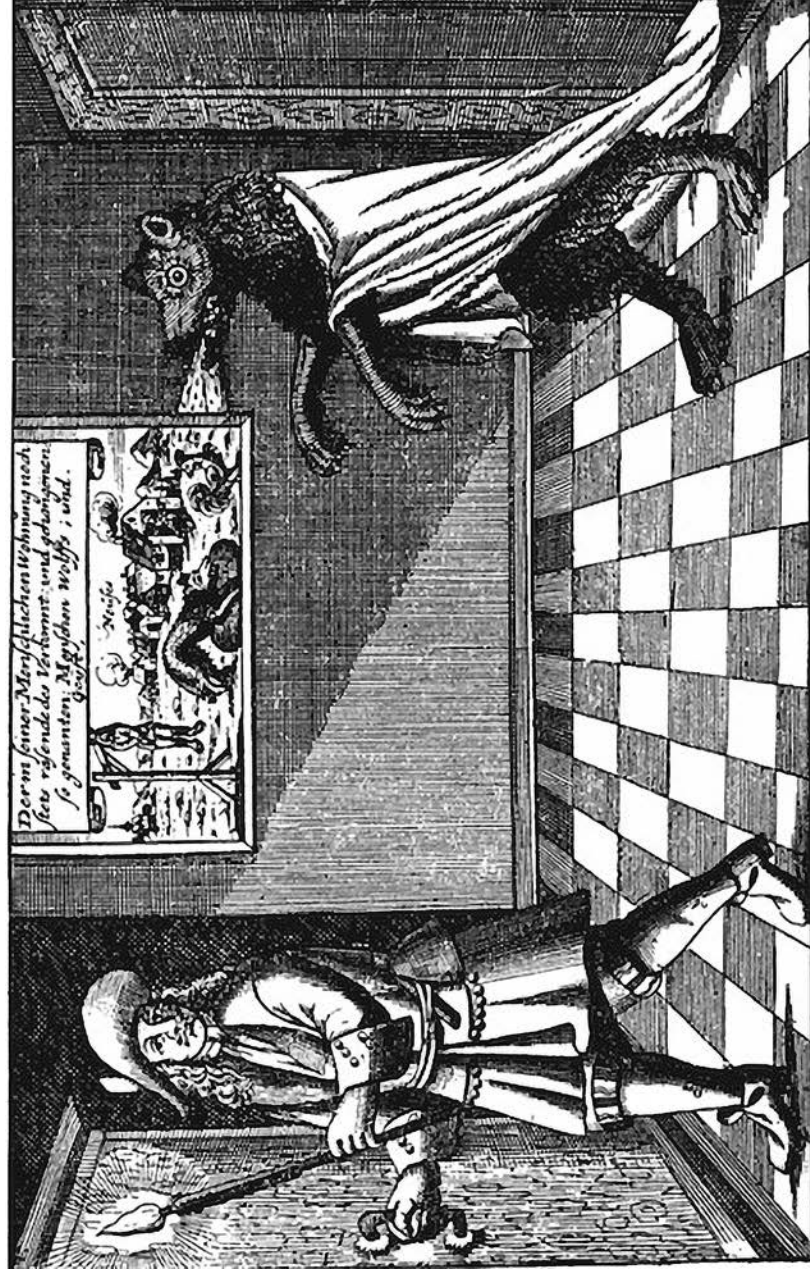
Аналогичный ответ применим к аргументу, что самодокументированные программы требуют большего количества нажатий клавиш. В печатном документе требуется минимум одно нажатие на клавишу для каждого символа на каждый черновой экземпляр. Самодокументирующаяся программа имеет меньшее суммарное число символов, а также меньше нажатий на клавиши, так как черновики не перепечатываются.

Как насчет блок-схем и структурных графов? Если используется только структурный граф самого высокого уровня, то он может безопасно храниться как отдельный документ, поскольку не подвержен частым изменениям. Но он, безусловно, может быть встроен в исходную программу в качестве комментария, и это кажется разумным.

В какой степени используемая выше методика применима к программам на языке ассемблера? Думаю, что базовый подход самодо-

кументирования полностью применим. Пространство и форматы менее свободны и поэтому не могут быть так же гибко использованы. Имена и структурные объявления, безусловно, могут быть использованы. Макросы могут очень сильно помочь. Широкое использование комментариев к абзацам является хорошей практикой на любом языке.

Но подход к самодокументированию стимулируется использованием языков высокого уровня и находит свою наибольшую силу и свое наибольшее оправдание в языках высокого уровня, используемых с онлайн-системами, будь то пакетные или интерактивные. Как я уже говорил, такие языки и системы очень сильно помогают программистам. Поскольку машины сделаны для людей, а не люди для машин, их использование оправданно как с экономической точки зрения, так и с человеческой.



Оборотень из Эшенбаха. Германия, штриховая гравюра, 1685. Предоставлено коллекцией
Грейнджера, Нью-Йорк

СЕРЕБРЯНОЙ ПУЛИ НЕТ — СУЩЕСТВЕННЫЕ И ЧАСТНЫЕ ПРИЗНАКИ ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Ни в технологии, ни в методике менеджмента нет ни одной разработки, которая сама по себе обещала бы повысить хоть на один порядок продуктивность, надежность и простоту в течение десятилетия.

АННОТАЦИЯ¹

Работа по созданию любого крупного ПО включает существенные задачи — конструирование сложных концептуальных структур, которые составляют суть программы, и частные (второстепенные) задачи — представление этих абстрактных сущностей на языке программирования с дальнейшим отображением этого в машинный код, с учетом ограничений скорости и пространства. Большинство крупных прошлых достижений в продуктивности программного обеспечения получены в результате устранения искусственных барьеров, которые чрезмерно затрудняли решение частных задач, таких как жесткие аппаратные ограничения, неудобные языки программирования, нехватка машинного времени. Сколько из того, что инженеры программного обеспечения делают сейчас, по-прежнему посвящено частным признакам, а не ключевым?

Если только этот объем не составляет более девяти десятых всех усилий, то сжатие всех частных видов деятельности до нуля времени не даст улучшения на порядок.

Поэтому, похоже, настало время обратиться к существенным задачам программирования, связанным с моделированием концептуальных структур большой сложности. Я предлагаю:

- Использовать то, что доступно на массовом рынке, во избежание разработки того, что можно купить.
- Использовать быстрое прототипирование как часть запланированной итерации по установлению требований к программному обеспечению.
- Нарастивать программное обеспечение органично, добавляя в системы все больше и больше функций по мере их выполнения, использования и тестирования.
- Выявлять и растить великих разработчиков нового поколения.

ВВЕДЕНИЕ

Из всех монстров, которые наполняют кошмары нашего фольклора, ни один не пугает больше, чем оборотни, потому что что-то привычное неожиданно превращается в ужасных страшил. Для них мы ищем серебряные пули, которые могут волшебным образом отправить их на покой.

Хорошо знакомый проект по разработке программного обеспечения чем-то похож на оборотня (по крайней мере, с точки зрения управленца) — обычно невинный и простой, но способный обернуться монстром из-за дедлайнов, раздувшихся бюджетов и неработающих продуктов. И поэтому мы слышим отчаянные крики о серебряной пуле, о чем-то, что в мгновение ока понизит стоимость программного обеспечения, так же быстро, как это происходит со снижением стоимости компьютерного железа.

Но когда мы смотрим на горизонт грядущего десятилетия, мы не видим никакой серебряной пули. Ни в технологии, ни в методиках управления нет ни одной разработки, которая сама по себе обещала бы повысить хоть на один порядок продуктивность, надежность и простоту. В этой главе мы попытаемся понять, почему это так, рассмотрев как природу задачи разработки программного обеспечения, так и свойства предлагаемых пуль.

Однако скептицизм не есть пессимизм. Хотя мы не видим никаких поразительных прорывов и действительно считаем, что они несовместимы с природой программного обеспечения, сейчас происходит много обнадеживающих инноваций. Дисциплинированные, последовательные усилия по их развитию, распространению и эксплуатации действительно должны привести к улучшению на порядок. Царского пути нет, но есть путь.

Первым шагом на пути к контролю заболеваний была замена демонических и гуморальных теорий микробной теорией. Этот шаг

стал основой и разбил все надежды на волшебные решения. Сам этот шаг, обещавший надежду, опроверг все мечты о чудесном исцелении. Он подсказал исследователям, что прогресс будет осуществляться шажками, с большим трудом, и что постоянное и неослабное внимание нужно уделять санитарии. Так же обстоит дело и с инженерией программного обеспечения сегодня.

ДОЛЖНО ЛИ БЫТЬ ТЯЖЕЛО? СУЩЕСТВЕННЫЕ ТРУДНОСТИ

Мало того что сейчас в поле зрения нет серебряных пуль, так еще и сама природа программного обеспечения делает их появление маловероятным — никакое изобретение не сделает для продуктивности, надежности и простоты программного обеспечения то, что электроника, транзисторы и крупномасштабная интеграция сделали для компьютерного оборудования. Двукратный рост каждые два года для нас пока недостижим.

Во-первых, мы должны заметить, что аномалия заключается не в том, что прогресс программного обеспечения идет так медленно, а в том, что прогресс компьютерного оборудования идет так быстро. Ни одна другая технология с тех пор, как зародилась цивилизация, за 30 лет не добилась выигрыша в цене-производительности на шесть порядков. Ни в одной другой технологии нельзя выбрать выигрыш либо в повышении производительности, либо в снижении затрат. Эти выигрыши вытекают из трансформации компьютерного производства из сборочной отрасли в обрабатывающую.

Во-вторых, для того чтобы увидеть, какие темпы прогресса мы можем ожидать в области программных технологий, давайте рассмотрим ее трудности. По Аристотелю, их можно разделить на *существенное* — трудности, присущие природе программного обеспечения, и *частности* — те трудности, которые сегодня сопровождают его производство, но которые ему не присущи.

О частных признаках я расскажу в следующем разделе. Сначала рассмотрим существенные признаки.

Существенный признак программной единицы — это конструкт из взаимосвязанных концепций: наборов данных, связей между элементами данных, алгоритмов и вызовов функций. Указанный существенный признак является абстрактным в том смысле, что концептуальный конструкт (или понятийное уموпостроение) является одинаковым в условиях многих разных представлений. Тем не менее он является очень точным и насыщенным деталями.

Я полагаю, что трудной частью разработки программного обеспечения является спецификация, дизайн и тестирование этого концептуального конструкта, а не работа по его представлению и тестированию верности представления. Конечно, мы по-прежнему допускаем синтаксические ошибки, но они являются незначительными по сравнению с концептуальными ошибками в большинстве систем.

Если это правда, то создавать программное обеспечение будет трудно всегда. Серебряной пули, в сущности, нет.

Рассмотрим неотъемлемые свойства этой несократимой сущности современных программных систем: сложность (complexity), подчиненность форме (conformity), изменчивость (changeability) и невидимость (invisibility).

СЛОЖНОСТЬ. Сложность программных объектов больше зависит от их размеров, чем, возможно, для любых других создаваемых человеком конструкций, поскольку никакие две их части не схожи между собой (по крайней мере, выше уровня операторов). Если они есть, мы сводим две подобные части в одну подпрограмму, открытую или закрытую. В этом отношении программные системы сильно отличаются от компьютеров, сооружений или автомобилей, где в изобилии встречаются повторяющиеся элементы.

Цифровые компьютеры сами по себе являются более сложными, чем большинство вещей, которые создают люди; они имеют очень большое количество состояний. Это затрудняет их понимание, описание и тестирование. Программные системы имеют на порядки больше состояний, чем компьютеры.

Схожим образом вертикальное масштабирование программной единицы — это не просто повторение одних и тех же элементов в большем размере; это обязательно увеличение числа разных элементов. В большинстве случаев элементы взаимодействуют друг с другом каким-то нелинейным образом, и сложность целого возрастает гораздо больше, чем линейно.

Сложность программного обеспечения является существенным признаком, а не частным. Отсюда описания программной единицы, абстрагируемые от ее сложности, часто абстрагируются от ее сути. Математика и физические науки добились больших успехов в течение трех столетий, построив упрощенные модели сложных явлений, выведя из них свойства и верифицируя их экспериментально. Это сработало, потому что особенности, игнорируемые в таких моделях, не были существенными признаками явлений. Это не работает, когда особенности являются существенным признаком.

Многие классические проблемы разработки программных продуктов вытекают из этой существенной сложности и ее нелинейных возражений вместе с размером.

Из сложности возникает трудность коммуникации между членами команды, что приводит к недостаткам продукта, перерасходам средств, срыву сроков. Из сложности возникает трудность перечисления, а тем более понимания всех возможных состояний программы, и отсюда возникает ненадежность. Из сложности функций возникает трудность вызова этих функций, что затрудняет использование программ. Из сложности структуры возникает

трудность расширения программ новыми функциями без создания побочных эффектов. Из сложности структуры возникают невизуализированные состояния, которые представляют собой ловушки безопасности.

Из сложности проистекают не только технические проблемы, но и проблемы управления. Эта сложность затрудняет общее видение, тем самым препятствуя концептуальной целостности. Она мешает отыскивать все свободные концы и контролировать их. Она создает огромное бремя усвоения и понимания, которое превращает текучку кадров в катастрофу.

ПОДЧИНЕННОСТЬ ФОРМЕ. Программисты не одиноки в столкновении со сложностью. Физика имеет дело с объектами чрезвычайной сложности даже на уровне элементарных частиц. Однако физик работает в твердой уверенности, что можно найти общие принципы, будь то кварки или общая теория поля. Эйнштейн неоднократно утверждал, что природа должна иметь простые объяснения, поскольку Богу не свойственны капризность и произвол.

Никакая такая вера не утешает инженера программного обеспечения. Большая часть сложности, которую он должен освоить, является привнесенной, навязанной без причины многими человеческими институтами и системами, которым его интерфейсы должны подчиняться. Они отличаются от интерфейса к интерфейсу и время от времени не из-за необходимости, а только потому, что они планировались разными людьми, а не Господом Богом.

Во многих случаях программное обеспечение должно подчиняться, потому что оно появилось совсем недавно. В других оно должно подчиняться, потому что воспринимается как наиболее подчиняемое. Но во всех случаях значительная часть сложности происходит от согласования с другими интерфейсами, и это невозможно упростить только в результате перепроектирования программного обеспечения.

ИЗМЕНЧИВОСТЬ. Программные объекты постоянно подвержены изменениям. Разумеется, как и сооружения, машины, компьютеры. Но произведенные вещи редко изменяются после производства; они заменяются более поздними моделями, или существенные изменения встраиваются в более поздние серийные копии того же самого базового дизайна. Отзывы автомобилей действительно случаются нечасто; полевые изменения компьютеров также редки. Но и то и другое встречается гораздо реже, чем модификации используемого программного обеспечения.

Отчасти это связано с тем, что программное обеспечение в системе воплощает ее функциональность, а функциональность является той частью, которая больше всего чувствительна к изменениям. Как правило, программное обеспечение легче изменить — это чистый мысленный материал, бесконечно податливый. Сооружения действительно меняются, но понятная всем высокая стоимость изменений служит лекарством от капризов изменяющих.

Все успешное программное обеспечение претерпевает изменение. При этом работают два процесса. Поскольку программный продукт оказывается полезным, люди пробуют его в новых случаях на границе или за пределами исходной предметной области. Давят, чтобы достичь введения расширенной функциональности, главным образом пользователи, которые любят базовую функциональность и изобретают для нее новые применения.

Во-вторых, успешное программное обеспечение также используется дольше срока жизни машины, для которой оно изначально было написано. Если не новые компьютеры, то по крайней мере новые диски, новые дисплеи, новые принтеры; и программное обеспечение должно быть приведено в соответствие своим новым носителям потенциала.

Одним словом, программный продукт встроен в культурную матрицу использования, пользователей, законов и машинных носителей.

Все они постоянно меняются, их изменения неумолимо заставляют меняться и программный продукт.

НЕВИДИМОСТЬ. Программное обеспечение является невидимым и невизуализируемым. Геометрические абстракции являются мощными инструментами. Поэтажный план сооружения помогает как архитектору, так и клиенту оценить пространство, потоки движения, виды. Противоречия становятся очевидными, упущения можно уловить. Масштабные чертежи механических деталей и фигурные модели молекул, хотя и абстрактные, служат той же цели. Геометрическая реальность запечатлена в геометрической абстракции.

Реальность программного обеспечения по своей сути не встроена в пространство. Следовательно, у него нет готового геометрического представления в том виде, в каком Земля имеет географические карты, кремниевые чипы имеют диаграммы, компьютеры имеют схемы соединений. Как только мы пытаемся построить диаграмму структуры программного обеспечения, мы обнаруживаем, что она представляет собой не один, а несколько общих ориентированных графов, наложенных один на другой. Несколько графов могут представлять поток управления, поток данных, шаблоны зависимостей, временную последовательность, связи пространства имен. Они обычно даже не плоские, а тем более не иерархические. Действительно, одним из способов установления концептуального контроля над такой структурой является принудительное сокращение связей до тех пор, пока один или несколько графов не станут иерархическими.²

Несмотря на прогресс в ограничении и упрощении структур программного обеспечения, они, в сущности, остаются невизуализируемыми, лишая ум некоторых из его наиболее мощных концептуальных инструментов. Этот недостаток не только затрудняет процесс дизайна внутри ума, но и серьезно затрудняет коммуникацию между умами.

ПРОШЛЫЕ ПРОРЫВЫ РЕШАЛИ ЧАСТНЫЕ ТРУДНОСТИ

Если мы рассмотрим три этапа в технологии программного обеспечения, наиболее плодотворные в прошлом, то обнаружим, что все они были сделаны в направлении решения различных крупных проблем разработки программ, но эти трудности были частными признаками, а не существенными. Можно также видеть естественные пределы экстраполирования каждого из этих направлений.

ЯЗЫКИ ВЫСОКОГО УРОВНЯ. Безусловно, самым мощным стимулом для повышения продуктивности, надежности и простоты программного обеспечения было широкое использование языков высокого уровня. Большинство наблюдателей приписывают этому развитию по крайней мере пятикратный рост продуктивности и сопутствующий рост надежности, простоты и понятности.

Чего достигает язык высокого уровня? Он освобождает программу от большей части ее привнесенной сложности. Абстрактная программа состоит из концептуальных конструкторов: операций, типов данных, последовательностей и связи. Конкретная машинная программа занята битами, регистрами, условиями, ветвлениями, каналами, дисками и т. д. В той мере, в какой язык высокого уровня воплощает конструкторы, необходимые в абстрактной программе, и избегает всех более низких, он устраняет целый уровень сложности, который никогда не был присущ программе вообще.

Самое большее, что может сделать язык высокого уровня, — это предоставить все конструкторы, которые программист мыслит в абстрактной программе. Конечно, уровень нашей изощренности в мышлении о структурах данных, типах данных и операциях неуклонно растет, но с постоянно убывающей скоростью. И развитие языка приближается к изощренности пользователей.

Более того, в какой-то момент усовершенствование языков высокого уровня становится тяжелым бременем, которое увеличивает, а не уменьшает сложность интеллектуальной задачи для пользователя, редко использующего эзотерические конструкции.

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ВРЕМЕНИ. Большинство наблюдателей связывают введение совместного использования машинного времени (или разделение времени) с повышением продуктивности программистов и качества их продукта, хотя и не столь значительным, как это было достигнуто с помощью языков высокого уровня.

Совместное использование времени является совершенно иной трудностью. Совместное использование времени сохраняет непосредственность и, следовательно, позволяет нам поддерживать общее видение комплексности. Медленный оборот пакетного программирования означает, что мы неизбежно забываем мелочи, если не саму суть того, о чем мы думали, когда прекращали программирование и требовали компиляции и исполнения. Это прерывание сознания приводит к большим потерям времени, ибо мы должны освежать понимание. Самым серьезным последствием вполне может стать неполное понимание всего, что происходит в сложной системе.

Медленный оборот, как и сложность машинного языка, является скорее частной, чем существенной трудностью программного процесса. Пределы вклада совместного использования времени определяются непосредственно. Принципиальный эффект заключается в сокращении времени отклика системы. По мере приближения его к нулю оно переходит порог скорости человеческого восприятия, составляющей около 100 миллисекунд. За его пределами никаких преимуществ ожидать не приходится.

УНИФИЦИРОВАННЫЕ СРЕДЫ ПРОГРАММИРОВАНИЯ. Считается, что Unix и Interlisp, первые широко распространенные интегрированные

среды программирования, повысили производительность в несколько раз. Почему?

Они борются с частными трудностями *совместного* использования программ, предоставляя интегрированные библиотеки, унифицированные форматы файлов, каналы и фильтры. Как результат, концептуальные структуры, которые в принципе всегда могут вызывать, подпитывать и использовать друг друга, действительно легко могут делать это на практике.

Этот прорыв, в свою очередь, стимулировал развитие целых наборов инструментов, поскольку каждый новый инструмент мог быть применен к любым программам, использующим стандартные форматы.

В силу этих успехов среды программирования являются предметом многих современных исследований в инженерии программного обеспечения. Мы проанализируем их перспективы и ограничения в следующем разделе.

НАДЕЖДЫ НА СЕРЕБРО

Теперь давайте рассмотрим технические разработки, которые чаще всего предлагаются в качестве потенциальных серебряных пуль. Какие задачи они решают? Являются ли эти задачи существенными или это напоминание наших частных трудностей? Предлагают ли они революционные достижения или постепенные?

АДА И ДРУГИЕ ДОСТИЖЕНИЯ В ЯЗЫКАХ ВЫСОКОГО УРОВНЯ. Одной из наиболее расхваливаемых новейших разработок является язык программирования Ada, универсальный язык высокого уровня 1980-х годов. Ada действительно не только отражает эволюционные улучшения в концепциях языка, но и воплощает в себе свойства, способствующие современным дизайнерским и моду-

ляризационным концепциям. Пожалуй, бóльшим достижением является философия Ada, чем язык Ada, поскольку эта философия — философия модуляризации, абстрактных типов данных, иерархического структурирования. Язык Ada является, пожалуй, перенасыщенным, естественным продуктом процесса, в котором требования были превращены в основу его разработки. Это не является фатальным, поскольку подмножество рабочих словарей может решить задачу усвоения, а достижения в аппаратном обеспечении дадут нам дешевые MIPS (миллион инструкций в секунду) для оплаты затрат на компиляцию. Усовершенствование структуризации программных систем действительно является очень хорошим применением для возросших MIPS, которые можно купить за деньги. Операционные системы, громко осуждавшиеся в 60-х годах за дороговизну памяти и вычислений, оказались хорошим способом применения быстродействия и дешевой памяти, полученных в результате быстрого развития аппаратных средств.

Тем не менее Ada не окажется серебряной пулей, которая убивает монстра продуктивности программного обеспечения. В конце концов это всего лишь еще один язык высокого уровня, и самый большой выигрыш от таких языков возник из первого перехода, от привнесенных сложностей машины вверх к более абстрактной постановке пошаговых решений. Как только эти частные признаки будут устранены, оставшиеся будут меньше, и выигрыш от их устранения, несомненно, тоже будет меньше.

Предсказываю, что через десятилетие, когда оценят эффективность Ada, будет признан значительный вклад этого языка, но не благодаря какой-либо отдельной его возможности и даже не благодаря им всем вместе взятым. Не окажутся причиной улучшений и новые среды Ada. Наибольший вклад Ada будет заключаться в том, что переход на этот язык привел к обучению программистов современной методике разработки программного обеспечения.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. Многие прогрессивные студенты возлагают больше надежд на объектно-ориентированное программирование, чем на любые другие преходящие технические причуды.³ Я один из них. Марк Шерман (Mark Sherman) из Дартмута отмечает, что мы с осторожностью должны различать две отдельные идеи, которые идут под этим именем: абстрактные типы данных и иерархические типы, также именуемые *классами*. Концепция абстрактного типа данных заключается в том, что тип объекта должен определяться именем, множеством правильных значений и множеством правильных операций, а не структурой его хранения, которая должна быть скрыта. Примерами являются пакеты языка Ada (с приватными типами) или модули языка Modula.

Иерархические типы, такие как классы языка Simula-67, позволяют определять общие интерфейсы, которые могут дополнительно уточняться путем предоставления подчиненных типов. Эти два понятия являются ортогональными — могут существовать иерархии без сокрытия и сокрытие без иерархий. Обе концепции представляют собой реальные достижения в искусстве создания программного обеспечения.

Каждое из них устраняет еще одну частную трудность из процесса, позволяя дизайнеру выражать сущность своего дизайна без необходимости выражать большие объемы синтаксического материала, которые не добавляют никакого нового информационного содержания. Как для абстрактных типов, так и для иерархических типов результатом является устранение некой более высокоуровневой частной трудности и появление возможности выражения высокоуровневого дизайна.

Тем не менее такие усовершенствования в лучшем случае могут устранить все частные трудности при выражении дизайна. Существенна сложность самого проекта, на что решение таких задач никак не может повлиять. Объектно-ориентированное програм-

мирование сможет достичь увеличения на порядок только в том случае, если остающаяся сегодня в нашем языке программирования необязательная работа по спецификации типов сама по себе ответственна за 9/10 усилий, затрачиваемых на проектирование программного продукта. Я сомневаюсь в этом.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ. Многие ожидают, что достижения в искусственном интеллекте обеспечат революционный прорыв, который даст увеличение продуктивности и качества программного обеспечения на порядок.⁴ Я так не считаю. Для того чтобы увидеть почему, мы должны проанализировать, что подразумевается под «искусственным интеллектом», а затем посмотреть, как он применяется.

Парнас прояснил терминологический хаос:

Сегодня широко используются два совершенно разных определения ИИ.

ИИ-1: использование компьютеров для решения задач, которые ранее могли быть решены только с помощью человеческого интеллекта.

ИИ-2: использование определенного множества методик программирования, известных как эвристическое программирование, или программирование, основанное на правилах. В этом подходе люди-эксперты являются предметом изучения и определяется то, какие эвристики или эмпирические правила они используют в решении задач. ...Программа корректируется для решения задач так, как, по-видимому, ее решает человек.

У первого определения скользкий смысл... Сегодня что-то может укладываться в определение ИИ-1, но как только мы увидим, как работает программа и поймем задачу, мы больше не будем думать об этом как об ИИ... К сожалению, я не могу идентифицировать совокупность технологий, уникальных для этой области... Большинство работы является специфичной для задачи, и для того, чтобы понять, как ее передать, требуется некоторая абстракция или творческий подход.⁵

Я полностью согласен с этой критикой. Методы, используемые для распознавания речи, по-видимому, имеют мало общего с методами, используемыми для распознавания образов, и оба они отличаются от методов, используемых в экспертных системах. Мне трудно понять, как распознавание образов, например, будет иметь какое-либо заметное значение в практике программирования. То же самое относится и к распознаванию речи. Самое сложное в разработке программного обеспечения — это решить, что говорить, а не просто говорить. Никакое содействие самовыражению не может дать больше, чем едва заметные выгоды.

ЭКСПЕРТНЫЕ СИСТЕМЫ. Наиболее продвинутой и наиболее широко применяемой частью ремесла искусственного интеллекта является технология сборки экспертных систем. Многие исследователи программного обеспечения усердно работают, применяя эту технологию к среде сборки программного обеспечения.⁶ Какова концепция и каковы перспективы этой технологии?

Экспертная система — это программа, содержащая обобщенный генератор выводов и базу правил, предназначенную для приема входных данных и допущений и исследования логических следствий через заключения, выводимые из базы правил, предоставляющая заключения и рекомендации и предлагающая пользователю объяснение полученных результатов путем обратного прослеживания своих рассуждений. Помимо чисто детерминированной логики, генератор выводов обычно может работать с нечеткими или вероятностными данными.

Такие системы предлагают некоторые явные преимущества перед запрограммированными алгоритмами с целью получения одних и тех же решений одних и тех же задач:

- Технология механизмов логического вывода разрабатывается независимым от приложения (прикладной области) способом,

а затем применяется во многих приложениях. Действительно, эта технология хорошо развита.

- Изменяемые части специфичных для приложения материалов кодируются в базе правил единообразным образом, и предоставляются инструменты для разработки, изменения, тестирования и документирования базы правил. Это упорядочивает большую часть сложности самого приложения.

Эдвард Фейгенбаум (Edward Feigenbaum) говорит, что сила таких систем исходит не от все более причудливых механизмов логического вывода, а, скорее, от все более насыщенных баз знаний, которые точнее отражают реальный мир. Я считаю, что наиболее важным достижением, предлагаемым указанной технологией, является отделение сложности приложения от самой программы.

Как это можно применить к работе по созданию программного обеспечения? Во многих отношениях: предлагая интерфейсные правила, консультируя по стратегиям тестирования, запоминая частоты типов ошибок, предлагая подсказки по оптимизации и т. д.

Возьмем, например, воображаемого советника по тестированию. В зачаточной форме диагностическая экспертная система весьма напоминает памятку пилота, по сути делая предположения относительно возможных причин затруднений. По мере развития базы правил рекомендации становятся более специфичными, принимая во внимание более изощренные симптомы проблемы, о которых сообщается. Можно представить такого помощника предлагающим сначала самые общие решения, но, по мере воплощения в базе правил все большей части структуры системы, становящегося все более разборчивым в генерируемых гипотезах и предлагаемых тестах. Такая экспертная система может самым радикальным образом отличаться от традиционных тем, что ее база правил, вероятно, должна быть иерархически модуляризована таким же образом, как

и соответствующий программный продукт. По мере того как этот продукт модульно модифицируется, база диагностических правил тоже может модифицироваться модульно.

Работа, необходимая для генерирования диагностических правил, — это работа, которая так или иначе должна быть выполнена при создании набора тестовых случаев для модулей и для системы. Если это делается в подходящей общей манере, с единообразной структурой правил и хорошим механизмом логического вывода, то она может фактически уменьшить общую трудоемкость генерирования тестовых примеров, а также помочь в пожизненном сопровождении и тестировании ее изменений. Точно так же мы можем постулировать других советников — вероятно, многих из них и, вероятно, простых — для других частей работы по созданию программного обеспечения.

Многие трудности стоят на пути ранней реализации полезных советников для разработчика программ. Важнейшей частью нашего воображаемого сценария является разработка простых способов перехода от спецификации структуры программы к автоматической или полуавтоматической генерации диагностических правил. Еще более трудной и важной является двойная работа приобретения знаний: отыскание аккуратных, умеющих проанализировать свою собственную работу экспертов, которые знают, почему они делают так, а не иначе; и разработка эффективных методов извлечения того, что они знают, и перегонки этого в базы правил. Необходимым условием для сборки экспертной системы является наличие эксперта.

Экспертные системы, безусловно, сослужат хорошую службу неопытному программисту. Накопленная мудрость лучших программистов, их опыт — это немалый вклад. Разрыв между лучшей практикой инженерии ПО и обычной практикой очень широк — возможно, шире, чем в любой другой инженерной дисциплине. Значение будет иметь инструмент распространения надлежащей практики.

«АВТОМАТИЧЕСКОЕ» ПРОГРАММИРОВАНИЕ. Почти 40 лет люди ждут и пишут об «автоматическом программировании» — генерации решающей задачу программы, исходя из формулировки спецификации этой задачи. Некоторые сегодня пишут так, как будто они ожидали, что эта технология обеспечит следующий прорыв.⁷

Парнас предполагает, что этот термин используется для красоты, а не для семантического содержания:

Одним словом, автоматическое программирование всегда было эвфемизмом для программирования на языке более высокого уровня, чем доступный программисту в данный момент инструмент.⁸

По существу, он утверждает, что в большинстве случаев автоматическое программирование представляет собой метод решения, а не задачу, спецификация которой должна существовать.

Можно найти исключения. Методика сборки генераторов является очень мощной, и она рутинно используется с хорошим преимуществом в программах для сортировки. Некоторые системы интегрирования дифференциальных уравнений также допускают прямую спецификацию задачи. Указанная система оценивала параметры, выбирала из библиотеки методы решения и генерировала программы.

У этих применений есть свойства, благоприятствующие автоматизации:

- Задачи легко характеризуются относительно небольшим числом параметров.
- Существует целый ряд известных методов решения, обеспечивая библиотеки альтернатив.
- Обширный анализ привел к точным правилам отбора методики решения с учетом параметров задачи.

Едва ли возможно обобщение таких методов на весь мир обычных программных систем, в котором ситуация с такими приятными свойствами являются исключениями. Трудно даже представить себе, как мог бы произойти этот прорыв в обобщении.

ГРАФИЧЕСКОЕ ПРОГРАММИРОВАНИЕ. Любимым предметом кандидатских диссертаций в инженерии ПО является графическое, или визуальное, программирование, применение компьютерной графики к разработке программного обеспечения.⁹ Иногда перспективность такого подхода основываются на аналогии с разработкой микросхем VLSI (СБИС), где компьютерная графика играет столь большую роль. Иногда такой подход оправдывается рассмотрением блок-схем как идеальной среды дизайна программ и предоставлением мощных средств обеспечения для их разработки.

Ничего даже убедительного, а тем более захватывающего из этих усилий пока не вышло. Я убежден, что ничего и не будет.

Во-первых, как я уже как-то говорил, блок-схема является очень плохой абстракцией структуры программного обеспечения.¹⁰ Действительно, лучше всего рассматривать ее как попытку Беркса, фон Неймана и Гольдстейна обеспечить крайне необходимый язык управления высокого уровня для их предлагаемого компьютера. В том жалком виде — многие страницы соединенных линиями прямоугольников, — в котором сегодня разрабатываются блок-схемы, они доказали, в сущности, свою бесполезность: программисты рисуют их после, а не до создания описываемых ими программ.

Во-вторых, сегодняшние экраны имеют слишком мало пикселей, для того чтобы показать как масштаб, так и подробности любой серьезной детальной диаграммы программного обеспечения. Так называемая настольная метафора сегодняшней рабочей станции — это метафора кресла в самолете. Любой, кто перебирал стопку бумаг, сидя в кресле между двумя дородными пассажирами, сразу поймет разницу — одновременно можно увидеть лишь очень не-

многие вещи. Истинный рабочий стол обеспечивает общее видение и произвольный доступ к десятку страниц. Более того, когда приступы творчества становятся сильными, немало программистов или составителей документации, как известно, оставляют рабочий стол ради более просторной поверхности пола. Технология аппаратного обеспечения должна будет значительно продвинуться вперед, чтобы предоставляемый экранами обзор был достаточным для задач разработки ПО.

Что более фундаментально, как я аргументировал выше, — программное обеспечение очень трудно визуализировать. Независимо от того, строим ли мы диаграмму потока управления, вложенности области действия переменной, перекрестных ссылок между переменными, потока данных, иерархических структур данных или диаграмму чего-то еще, мы ощущаем только одну размерность запутанно взаимосвязанного программного слона. Если мы наложим все диаграммы, сгенерированные многими соответствующими представлениями, то будет трудно извлечь какое-либо глобальное общее видение. Аналогия VLSI является в корне ошибочной — дизайн микросхемы имеет вид слоистого двумерного объекта, геометрия которого отражает его сущность. А программная система таковой не является.

ВЕРИФИКАЦИЯ ПРОГРАММ. Большая часть усилий в современном программировании затрачивается на тестирование и исправление ошибок. Можно ли отыскать серебряную пулю, устранив ошибки в исходном коде, на этапе дизайна системы? Можно ли радикально повысить как продуктивность, так и надежность продукта, следуя совершенно иной стратегии доказательства правильности дизайнов, прежде чем огромные усилия будут направлены на их имплементацию и тестирование?

Я не верю, что нас ждет здесь волшебство. Верификация программ является очень мощной концепцией, и она будет очень важна для таких вещей, как безопасные ядра операционных систем. Технологи-

гия не обещает, однако, экономить рабочую силу. Верификации — это настолько большая работа, что всего лишь несколько солидных программ когда-либо были верифицированы.

Верификация программ не означает безошибочные программы. Здесь тоже нет никакого волшебства. Математические доказательства также могут быть ошибочными. Таким образом, в то время как верификация может уменьшить нагрузку, связанную с тестированием программы, она не может ее устранить.

Что более серьезно, даже совершенная верификация программы может только установить, что программа соответствует ее спецификации. Самая трудная часть работы по созданию программного обеспечения заключается в достижении полной и единообразной спецификации, и большая часть сущности разработки программы на самом деле заключается в отладке спецификации.

СРЕДЫ И ИНСТРУМЕНТЫ. Насколько больше пользы можно ожидать от прорывных исследований в области оптимальных сред программирования? Инстинктивно кажется, что задачи, которые сулили наибольшую отдачу, были в числе первых, за которые взялись, и их уже решили: иерархические файловые системы, единообразные форматы файлов для получения единообразных программных интерфейсов и обобщенных инструментов. Специфичные для языка умные редакторы являются новшеством, которое еще не нашло широкого применения на практике, но самое большее, что они обещают, — это свобода от синтаксических и простых семантических ошибок.

Возможно, самым большим достижением, которое еще предстоит реализовать в среде программирования, является использование интегрированных систем баз данных для отслеживания мириад деталей, которые должны точно воспроизводиться отдельным программистом и поддерживаться в актуальном состоянии группой сотрудников в одной системе.

Несомненно, эта работа стоит того, и она принесет определенные плоды как в продуктивности, так и в надежности. Но по самой своей природе возврат от вложений уже не будет существенным.

РАБОЧИЕ СТАНЦИИ. Какие выгоды можно ожидать для разработки программного обеспечения от определенного и быстрого увеличения мощности и объема памяти отдельной рабочей станции? И сколько MIPS можно задействовать плодотворно? Составление и редактирование программ и документов полностью поддерживается современными скоростями. Компиляция может быть существенно ускорена, но и десятикратный рост скорости компьютеров все еще оставит размышление доминирующей деятельностью в рабочем дне программиста. В самом деле, теперь это, кажется, именно так.

Более мощные рабочие станции мы, безусловно, приветствуем. Но волшебных улучшений от них мы ожидать не можем.

МНОГООБЕЩАЮЩИЕ АТАКИ НА КОНЦЕПТУАЛЬНУЮ СУЩНОСТЬ

Несмотря на то что ни один технологический прорыв не обещает дать волшебных результатов, с которыми мы так хорошо знакомы в области аппаратного обеспечения, в настоящее время часто мы имеем как хорошо сделанную работу, так и обещание устойчивого, хотя и незаметного прогресса. Все технологические атаки на частные признаки процесса программного обеспечения принципиально ограничены уравнением продуктивности:

$$\text{Время выполнения работы} = \sum_i (\text{Частота})_i \times (\text{Время})_i.$$

Если, как я полагаю, концептуальные составляющие задачи сейчас отнимают большую часть времени, то никакая работа над составными частями задачи, являющимися просто выражением концепций, не даст большого выигрыша.

Следовательно, мы должны рассмотреть те направления, которые затрагивают сущность задачи сборки ПО, формулировку этих комплексных концептуальных структур. К счастью, некоторые из них являются очень многообещающими.

ПОКУПКА ПРОТИВ РАЗРАБОТКИ. Самое радикальное решение из возможных для разработки программного обеспечения состоит в том, чтобы вообще отказаться от его создания.

С каждым днем делать это становится все проще, поскольку все больше и больше поставщиков предлагают все более совершенные программные продукты для головокружительного разнообразия приложений. В то время как мы, инженеры программного обеспечения, работали над методикой производства, революция персональных компьютеров создала не один, а много массовых рынков для программного обеспечения. В каждом газетном киоске выставлены ежемесячные журналы, разложенные по типам компьютеров, которые рекламируют и рассматривают десятки продуктов по ценам от нескольких долларов до нескольких сотен долларов. Более специализированные источники предлагают очень мощные продукты для рабочих станций и других рынков UNIX. Даже инструменты и среды для программирования могут быть приобретены прямо с полки. Я где-то даже предложил рыночную площадку для отдельных модулей.

Любой такой продукт дешевле купить, чем создать заново. Даже при стоимости в \$100 000 купленная часть программного обеспечения равноценна годовому окладу одного программиста. И поставка осуществляется немедленно! По крайней мере, для реально существующих продуктов, просpekt которых разработчик может

послать счастливому пользователю. Кроме того, такие продукты, как правило, гораздо лучше документированы и несколько лучше сопровождаются, чем доморощенное программное обеспечение.

Развитие массового рынка — это, я считаю, самый глубокий долгосрочный тренд в инженерии программного обеспечения. Стоимость программного обеспечения всегда была стоимостью разработки, а не стоимостью создания его копий. Распределение этой стоимости даже среди нескольких пользователей радикально сокращает затраты на одного пользователя. Еще один подход заключается в том, что использование n копий программной системы фактически умножает продуктивность ее разработчиков на n . То есть является ростом продуктивности индустрии и профессии.

Ключевым вопросом, конечно, является применимость. Могут ли я использовать имеющийся типовой программный продукт для выполнения своей работы? Здесь произошла удивительная вещь. В течение 1950-х и 1960-х годов исследования за исследованиями показывали, что пользователи не будут использовать типовые программы для расчета заработной платы, управления запасами, дебиторской задолженности и т. д. Требования были слишком узкоспециализированными, а вариации от случая к случаю — слишком высокими. В течение 1980-х годов мы находим такие программы высоковоластовыми и широко используемыми. Что же изменилось?

Едва ли сами программы. Они могут быть несколько более обобщенными и несколько более настраиваемыми под клиента, чем раньше, но ненамного. И едва ли способ использования этих программ. Во всяком случае потребности бизнеса и науки сегодня разнообразнее и сложнее, чем 20 лет назад.

Резко изменилось соотношение стоимости компьютеров и программ. Покупатель машины по цене \$2 миллиона в 1960 году чувствовал, что может позволить себе еще \$250 тысяч на при-

способленную под свои нужды программу расчета заработной платы, которая легко и незаметно проскальзывает во враждебную компьютеру социальную среду. Покупатели офисных машин по цене \$50 тысяч сегодня не могут себе позволить заказные программы расчета заработной платы; поэтому они адаптируют свои процедуры расчета заработной платы к имеющимся программам. Компьютеры сейчас настолько распространены, если еще не настолько любимы, что адаптация воспринимается как нечто само собой разумеющееся.

В моем аргументе есть исключения, которые состоят в том, что за эти годы обобщение пакетов программного обеспечения мало изменилось: электронные таблицы и простые системы баз данных. Эти мощные инструменты, столь очевидные в ретроспективе и все же столь поздно появляющиеся, поддаются бесчисленному множеству применений, часть из которых являются совершенно неортодоксальными. Статьи и даже книги теперь изобилуют советами, как справляться с неожиданными задачами с помощью электронной таблицы. Огромная масса приложений, которые ранее были написаны как заказные программы на Cobol или генераторе отчетов Report Program Generator, теперь шаблонно выполняются с помощью этих инструментов.

Многие пользователи теперь управляют своими компьютерами изо дня в день, работая с различными приложениями, ни разу не написав программы. Действительно, многие не могут писать новые программы для своих машин, но они, тем не менее, искусны в решении новых задач с помощью них.

Считаю, что самая мощная стратегия повышения продуктивности использования программного обеспечения для многих организаций сегодня состоит в оснащении работников интеллектуального труда, так часто критикуемых за неумение использовать компьютеры, персональными компьютерами и хорошими текстовыми и графическими редакторами, программами работы с файлами

и электронными таблицами. Та же самая стратегия, с обобщенными математическими и статистическими пакетами и некоторыми простыми возможностями программирования, также будет работать для сотен лабораторных ученых.

УТОЧНЕНИЕ ТРЕБОВАНИЙ И БЫСТРОЕ ПРОТОТИПИРОВАНИЕ. Самая трудная часть разработки программной системы заключается в том, чтобы точно решить, что разрабатывать. Никакая другая часть концептуальной работы не является такой трудной, как установление подробных технических требований, включая все интерфейсы для людей, машин и других программных систем. Никакая другая часть работы так не калечит результирующую систему, если она сделана неправильно. Никакая другая часть не является более трудной в исправлении позже.

Поэтому наиболее важной функцией, которую создатели программного обеспечения выполняют для своих клиентов, является итеративное извлечение и уточнение требований к продукту. По правде говоря, клиенты сами не знают, чего хотят. Обычно они не знают, на какие вопросы нужно дать ответ, и почти никогда не задумывались над задачей настолько детально, как это нужно указать в спецификации. Даже простой ответ: «сделать так, чтобы новая программная система работала как наша старая ручная система обработки информации» — на самом деле является слишком простым. Клиенты никогда не хотят именно этого. Более того, комплексные программные системы являются тем, что действует, что движется, что работает. Динамику этого действия трудно себе представить. Поэтому при планировании любой программной деятельности необходимо оставить резерв для взаимодействия между клиентом и разработчиком как составную часть определения системы.

Я бы пошел еще дальше и заявил, что для клиентов, даже для тех, кто работает с программистами, действительно невозможно полностью и правильно определить точные требования к современному

продукту до того, как будет построено и испытано несколько версий продукта, спецификации к которому они составляют.

В задаче сборки программного обеспечения одним из наиболее многообещающих современных технологических усилий, которое затрагивает существенные, а не случайные признаки, является разработка подходов и инструментов для быстрого прототипирования систем как части итеративного процесса разработки спецификаций.

Прототип программной системы — это такая система, которая симулирует важные интерфейсы и выполняет главные функции предполагаемой системы, не будучи обязательно связанной теми же аппаратными ограничениями по скорости, размеру или стоимости. Прототипы, как правило, выполняют типовые задачи приложения, но не пытаются обрабатывать исключения, правильно откликаться на недопустимые входные данные, корректно прерывать задачу и т. д. Цель прототипа — сделать реальной заданную концептуальную структуру, с тем чтобы клиент мог проверить ее на непротиворечивость и удобство использования.

Большинство современных процедур приобретения программного обеспечения основывается на допущении, что можно специфицировать удовлетворительную систему заранее, получить заявки на ее разработку, собрать ее и установить. Я думаю, что это допущение в корне неверно и что многие проблемы с приобретением программного обеспечения возникают из-за этой ошибки. Следовательно, они не могут быть исправлены без фундаментальной ревизии подхода, которая предусматривает итеративную разработку и спецификацию прототипов и продуктов.

ИНКРЕМЕНТАЛЬНОЕ РАЗВИТИЕ — наращивать, а не создавать программное обеспечение. Я до сих пор помню удар, который получил в 1958 году, когда впервые услышал, как товарищ говорит о *конструиро-*

вании программы, а не о ее *написании*. В мгновение ока он расширил мое представление о программном процессе. Метафорический сдвиг был мощным и точным. Сегодня мы понимаем, насколько похожи другие строительные процессы на разработку программного обеспечения, и мы свободно используем другие элементы метафоры, такие как *спецификации*, *сборка компонентов* и *возведение строительных лесов*.

Строительная метафора изжила себя. Пришло время снова изменить ее. Если, как я полагаю, сложность концептуальных структур, которые мы разрабатываем сегодня, не позволяет их точно специфицировать заранее и создавать безошибочно, то мы должны принять радикально иной подход.

Давайте обратимся к природе и изучим сложность живых существ, а не только бездушных творений человека. Здесь мы находим конструкторы, сложность которых вызывает у нас благоговейный трепет. Один только мозг невообразимо сложен, возможности его не поддаются имитации, он насыщен разнообразием, самозащитой и самообновлением. Секрет в том, что он растет, а не строится.

Так должно быть и с нашими программными системами. Несколько лет назад Харлан Миллс предложил, чтобы любая программная система развивалась постепенно.¹¹ То есть система должна быть сначала запущена, даже если она не делает ничего полезного, кроме вызова соответствующего множества фиктивных подпрограмм. Затем, шаг за шагом, она конкретизируется, а подпрограммы, в свою очередь, развиваются в действия или вызовы пустых заглушек уровнем ниже.

Я стал свидетелем самых впечатляющих результатов, когда начал продвигать этот метод среди разработчиков проектов на занятиях в моей Лаборатории инженерии ПО. Ничто за последнее десяти-

ление так радикально не изменило мою собственную практику или ее эффективность. Такой подход требует нисходящего дизайна, поскольку он представляет собой наращивание программного обеспечения сверху вниз. Он позволяет легко откатывать систему к предыдущему состоянию. Он способствует раннему прототипированию. Каждая добавленная функция и новое обеспечение для более сложных данных или обстоятельств органически вырастают из того, что уже есть.

Воздействие на моральный дух ошеломительное. Энтузиазм возрастает, когда есть работающая система, даже простая. Усилия удваиваются, когда на экране появляется первое изображение из новой графической программной системы, даже если это всего лишь прямоугольник. На каждом этапе процесса всегда есть работающая система. Я обнаружил, что команды способны за четыре месяца *вырастить* гораздо более сложные сущности по сравнению с теми, которые они могут *сконструировать*.

Те же самые преимущества могут быть реализованы на крупных проектах, как и на моих малых.¹²

ВЕЛИКИЕ РАЗРАБОТЧИКИ. Центральный вопрос о том, как улучшить программное ремесло, концентрируется, как всегда, на людях.

Мы можем получить хорошие проекты, следуя хорошим практикам вместо плохих. Хорошие практики разработки могут преподаваться. Программисты являются одним из самых интеллектуальных слоев населения, поэтому они легко усвоят хорошие практики. Отсюда и упор в США на распространение надлежащей современной практики. Новые учебные программы, новая литература, новые организации, такие как Институт инженерии программного обеспечения, — все это появилось для того, чтобы поднять уровень нашей практики от слабого до хорошего. И это совершенно правильно.

Тем не менее я не верю, что мы можем сделать следующий шаг вперед таким же образом. В то время как разница между слабыми концептуальными дизайнами и хорошими может лежать в здравости метода разработки, разница между хорошими дизайнами и великими, конечно, не в этом. Великие дизайны приходят от великих разработчиков. Создание программного обеспечения является *творческим* процессом. Здравая методика может наделить силой и освободить творческий ум; однако она не может воспламенить или вдохновить работника.

Различия не являются незначительными — они, скорее, похожи на Сальери и Моцарта. Исследование за исследованием показывает, что самые лучшие разработчики порождают структуры, которые быстрее, меньше, проще, чище и производятся с меньшими усилиями. Великий и средний различаются почти на порядок.

Небольшой ретроспективный анализ показывает, что, хотя многие прекрасные, полезные системы были спланированы комитетами и построены на основе проектов из многих звеньев, те программные системы, которые впечатлили страстных поклонников, являются продуктами одного или нескольких дизайнерских умов, великих разработчиков. Возьмем UNIX, APL, Pascal, Modula, интерфейс Small talk, даже Fortran и сопоставим с Cobol, PL/1, Algol, MVS/370 и MS-DOS (рис. 16.1).

Поэтому хоть я поддерживаю, причем решительно, осуществляемые в настоящее время усилия по распространению технологий и разработке учебных программ, я думаю, что самое важное, что мы можем предпринять, — это разработать способы воспитания великих разработчиков.

Ни одна программная организация не может игнорировать эту трудность. Хорошие менеджеры, какими бы редкими они ни были, не более редки, чем хорошие разработчики.

Да	Нет
UNIX	Cobol
APL	PL/I
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

Рис. 16.1. Впечатляющие продукты

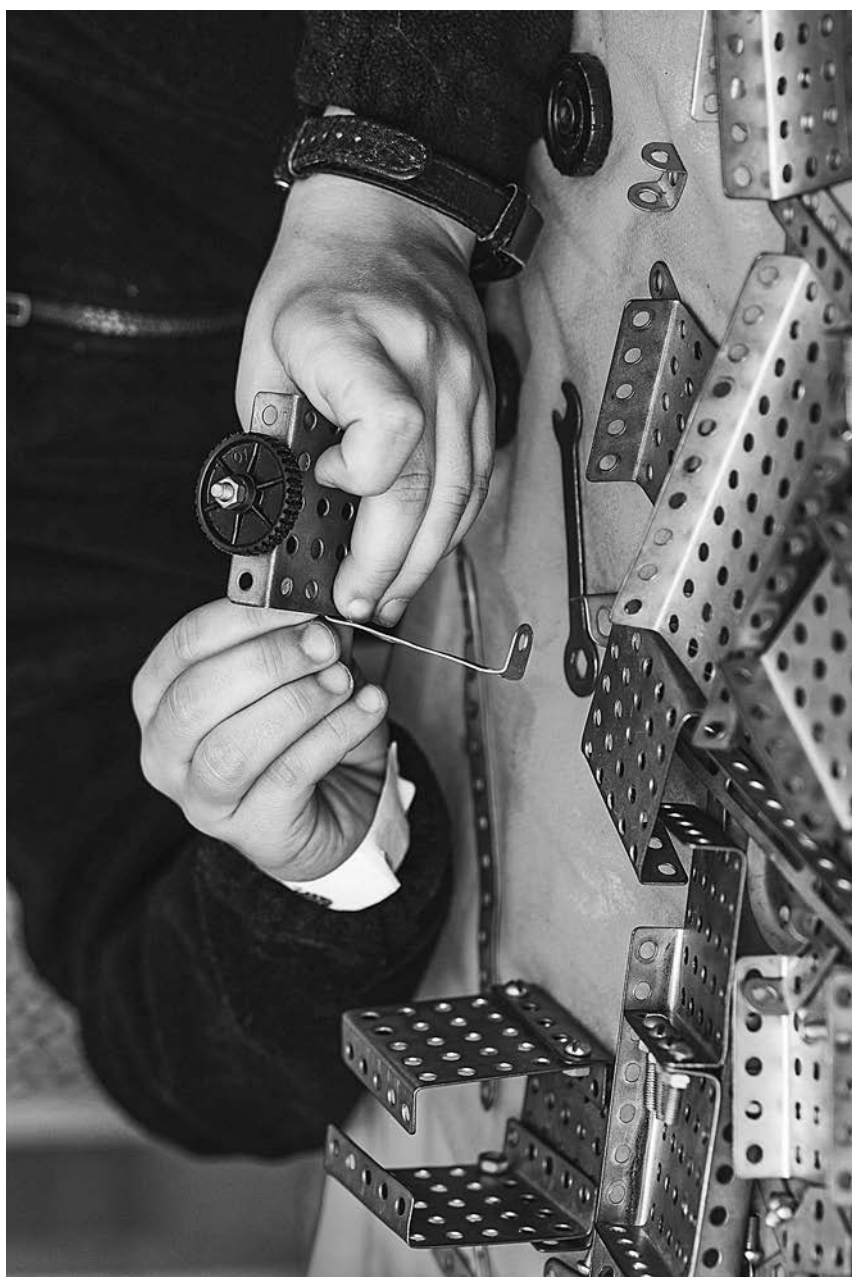
Великие дизайнеры и великие менеджеры встречаются очень редко. В большинстве организаций значительные усилия тратятся на поиск и выращивание подающих надежды менеджеров; я не знаю ни одной, которая затрачивает такие же усилия на поиск и подготовку великих разработчиков, от которых в конечном счете будет зависеть техническое совершенство продуктов.

Мое первое предложение состоит в том, что каждая организация должна определить и провозгласить, что великие разработчики имеют такую же важность для ее успеха, как и великие менеджеры, и что они должны воспитываться и вознаграждаться одинаково. Не только зарплата, но и привилегии признания — размер кабинета, мебель, личное техническое оснащение, дорожные фонды, поддержка персоналом — должны быть полностью эквивалентными.

Как вырастить великих разработчиков? Размер эссе не позволяет вдаваться в долгие обсуждения, но некоторые шаги являются очевидными:

- Систематически выявлять лучших разработчиков как можно раньше. Лучшие часто не самые опытные.
- Назначить карьерного наставника, который будет отвечать за подготовку перспективного разработчика и вести тщательное карьерное досье.

- Разработать и поддерживать план развития карьеры для каждого перспективного разработчика, включая тщательно подобранные стажировки ведущих разработчиков, эпизоды продвинутого формального образования и короткие курсы, вместе с индивидуальными заданиями и техническому лидерству.
- Обеспечить возможности для взаимодействия и взаимного стимулирования растущих разработчиков.



ПОВТОРНЫЙ ВЫСТРЕЛ «СЕРЕБРЯНОЙ ПУЛИ НЕТ»

У всякой пули — свое предназначение.

Вильгельм III, принц Оранский

Кто хочет увидеть образец совершенства, тот мечтает о том, чего никогда не было, нет и не будет.

Александр Поуп. Эссе о критике

ОБ ОБОРОТНЯХ И ДРУГИХ ЛЕГЕНДАРНЫХ МОНСТРАХ

Статья «Серебряной пули нет — существенные и частные признаки инженерии программного обеспечения» (теперь глава 16) изначально являлась заказным докладом на конференции IFIP '86 в Дублине и была опубликована в ее материалах.¹ Журнал *Computer* перепечатал ее, украсив готической обложкой, иллюстрированной кадрами из таких фильмов, как «Лондонский оборотень» (*The Werewolf of London*).² Они также сделали пояснительную вставку «Убить оборотня», изложив (современную) легенду о том, что помогут только серебряные пули. Я не знал об этой вставке и иллюстрациях до момента публикации и совсем не ожидал, что серьезная техническая статья будет оформлена таким образом.

Однако редакторы *Computer* достигли желаемого эффекта, и многие, судя по всему, эту статью прочитали. Поэтому для той главы я выбрал еще одно изображение оборотня. Древнее изображение почти комического существа. Надеюсь, что эта менее яркая картинка будет иметь такой же благотворный эффект.

СЕРЕБРЯНАЯ ПУЛЯ ВСЕ-ТАКИ ЕСТЬ — И ВОТ ОНА!

В статье «Серебряной пули нет» утверждается и аргументируется, что ни одна разработка в инженерии программного обеспечения не даст скачка повышения продуктивности программирования на порядок в течение 10 лет (с момента публикации статьи в 1986 году). Сейчас мы находимся на девятом году в этом десятилетии, поэтому настало время увидеть, как это предсказание сбывается.

В то время как «Мифический человеко-месяц» породил частое цитирование и мало споров, статья «Серебряной пули нет» вы-

звала статьи с опровержениями и письма в редакции журналов, поток которых не прекратился и по сей день.³ Большинство из них атакуют центральный аргумент о том, что нет никакого волшебного решения, и мое четкое мнение состоит в том, что его и не может быть. Многие соглашались с большинством аргументов в «СПН», но затем продолжают утверждать, что для программного зверя, которого изобрел автор, на самом деле серебряная пуля существует. Перечитывая сегодня ранние отклики, не могу не отметить, что патентованные средства, столь энергично предлагавшиеся в 1986 и 1987 годах, не возымели эффекта, на который претендовали.

Я покупаю аппаратное и программное обеспечение в основном с помощью теста «счастливый пользователь» — беседы с добросовестными клиентами, которые пользуются продуктом и им довольны. Точно так же я с готовностью поверю, что серебряная пуля материализовалась, когда добросовестный независимый пользователь выйдет и скажет: «Я использовал эту методику, инструмент или продукт, и это дало мне десятикратное повышение продуктивности программного обеспечения».

Многие респонденты сделали обоснованные исправления или разъяснения. Некоторые предприняли точечный анализ и опровержение, за что я им благодарен. В этой главе я поделюсь улучшениями и рассмотрю опровержения.

НЕЯСНОЕ ИЗЛОЖЕНИЕ ВЛЕЧЕТ НЕПОНИМАНИЕ

Некоторые авторы показывают, что у меня не получилось изложить некоторые аргументы четко.

СЛУЧАЙНЫЙ ПРИЗНАК. Центральный аргумент статьи «СПН» изложен в аннотации к главе 16 ясно настолько, насколько я в состоянии это сформулировать. Некоторые, однако, были сбиты с толку термина-

ми «accident»* (случайный признак, авария) и «accidental» (случайный, аварийный), которые следуют древнему использованию, восходящему к Аристотелю. Под термином «accidental» я имел в виду не то, что *происходит неожиданно*, и не *несчастье*, а, скорее, нечто *побочное, инцидентное* или *сопутствующее*.

Я бы не стал принижать частные части сборки программного обеспечения. Вместо этого я следую английскому драматургу, автору детективных романов и теологу Дороти Сэйерс, видя, что вся творческая деятельность состоит из (1) формулирования концептуальных конструкторов, (2) имплементации в реальных средах и (3) взаимодействия с пользователями в реальных целях.⁵ Часть сборки программного обеспечения, которую я назвал *существенным* признаком, — это мысленное изготовление концептуального конструктора; часть, которую я назвал *частным* признаком, — это процесс его имплементации.

ПОИСК ИСТИНЫ. Мне (но, правда, далеко не всем) кажется, что истинность центрального аргумента сводится к вопросу: какая доля суммарного программного усилия связана с точным и упорядоченным представлением концептуального конструктора и какая доля является усилием мысленного изготовления конструкторов? Отыскание и исправление недостатков частично приходится на каждую долю, в зависимости от того, являются ли недостатки концептуальными, такими как неспособность распознать какое-либо исключение, либо репрезентативными, такими как ошибка указателя или ошибка выделения памяти.

Мое мнение, что частная, или репрезентативная, часть работы теперь составляет примерно половину или меньше от суммарного объема. Поскольку эта доля является вопросом факта, ее величину, в принципе, можно было бы установить путем измерения.⁶

* Во избежание той же путаницы термин переведен как «частный» признак. — *Примеч. ред.*

В противном случае моя оценка может быть скорректирована более информированными и более актуальными оценками. Примечательно, что никто из тех, кто писал публично или в частном порядке, не утверждал, что частная (репрезентативная) доля равна девяти десятым.

Статья «СПН» бесспорно утверждает, что если частная доля работы составляет меньше девяти десятых от общей, то сжатие ее до нуля (что *потребовало бы* волшебства) не даст повышения продуктивности на порядок. *Нужно* целиться в существенный признак.

С момента публикации статьи «СПН» Брюс Блум (Bruce Blum) привлек мое внимание к работам Герцберга (Herzberg), Мауснера (Mausner) и Сайдермана (Sayderman)⁷ 1959 года. Они считают, что продуктивность могут повысить мотивационные факторы. С другой стороны, факторы окружения, какими бы положительными они ни были, не дадут такого эффекта; но эти факторы могут снижать продуктивность, когда они являются отрицательными. В статье «СПН» утверждается, что значительный прогресс в программном обеспечении был связан с устранением негативных факторов: потрясающе неудобных машинных языков, пакетной обработки с длительным и обратными временами, слабых инструментов и серьезных ограничений по памяти.

Являются ли *существенные* трудности поэтому *безнадежными*? Превосходная статья Брэда Кокса (Brad Cox) 1990 года «Серебряная пуля существует» (There Is a Silver Bullet) красноречиво демонстрирует многоразовый, взаимозаменяемый компонентный подход как атаку на концептуальную сущность задачи.⁸ Я с энтузиазмом соглашусь.

Кокс, однако, неправильно понимает статью «СПН» в двух местах. Во-первых, он читает ее как утверждение, что трудности с программным обеспечением возникают «из-за некоторого недостатка в том, как программисты сегодня строят программное обеспече-

ние». Мой аргумент состоял в том, что существенные трудности присущи концептуальной комплексности программных функций, которые должны быть спланированы и построены в любое время, любым методом. Во-вторых, он (и другие) прочитал статью «СПН» как утверждение, что нет никакой надежды исправить существенные трудности сборки программного обеспечения. Это не входило в мои намерения. Изготовление концептуального конструкта действительно имеет в качестве неотъемлемых трудностей сложность, подчиненность форме, изменчивость и невидимость. Проблемы, вызванные каждой из этих трудностей, могут, однако, быть смягчены.

СЛОЖНОСТЬ РАЗДЕЛЯЕТСЯ НА УРОВНИ. Например, сложность является самой серьезной внутренне присущей трудностью, но не всякая сложность является неизбежной. Бóльшая часть, но не вся, концептуальной сложности наших программных конструктов проистекает из произвольной сложности самих приложений. Действительно, Ларс Седал (Lars Sedahl) из *MYSIGMA Sedahl and Partners*, международной консалтинговой компании, пишет:

По моему опыту, большинство сложностей, возникающих в работе систем, являются симптомами организационных сбоев. Попытка смоделировать эту реальность с помощью столь же комплексных программ на самом деле означает консервацию беспорядка вместо решения проблем.

Стив Лукашик (Steve Lukasik) из *Northrop* утверждает, что даже организационная сложность, возможно, не является произвольной, но может быть восприимчива к принципам упорядочения:

Я обучался физике и, таким образом, вижу «сложные» вещи как поддающиеся описанию в терминах более простых понятий. Возможно, вы правы — я не стану утверждать, что все сложные вещи поддаются упорядочиванию... По тем же правилам аргументации вы не можете утверждать, что они не могут.

...Вчерашняя сложность — это завтрашняя упорядоченность. Сложность молекулярного беспорядка уступила место кинетической теории газов и трем законам термодинамики. В силу сказанного, программное обеспечение, возможно, никогда не раскроет эти виды принципов упорядочения, но на вас лежит бремя, чтобы объяснить, почему нет. Это не проявление моей беспечности или желания поспорить. Я верю, что когда-нибудь «сложность» программного обеспечения будет понята в терминах нескольких понятий более высокого порядка (инвариантов для физика).

Я не предпринимал более глубокий анализ, к которому призывает Лукашик. Как дисциплина, мы нуждаемся в расширенной теории информации, которая квантифицирует информационное содержимое статических структур, так же как теория Шеннона делает это для каналов коммуникации. Это совершенно выше моего понимания. Лукашику я просто отвечаю, что сложность системы — это функция множества деталей, каждая из которых должна быть точно определена либо по какому-то общему правилу, либо деталь за деталью, но не только исключительно статистически. Представляется весьма маловероятным, что несогласованные работы многих умов должны обладать достаточной связностью, чтобы быть точно описанными общими правилами.

Большая часть сложности в программном конструкте, однако, существует не из-за подчиненности внешнему миру, а из-за самой реализации — ее структур данных, ее алгоритмов, ее связности. Нарращивание программного обеспечения более высокоуровневыми частями, построенными кем-то другим или используемыми повторно из собственного прошлого, позволяет избежать столкновения с целыми слоями сложности. Статья «СПН» выступает за искреннюю атаку на проблему сложности, вполне оптимистично полагая, что прогресс может быть достигнут. Она выступает за добавление необходимой сложности в программную систему:

- иерархически, модулями и объектами, организованными по слоям;
- инкрементально, так, чтобы система всегда работала.

АНАЛИЗ ХАРЕЛА

Дэвид Харел (David Harel) в статье 1992 года «Жаля серебряную пулю» (*Biting the Silver Bullet*) предпринимает самый тщательный анализ статьи «СПН», который когда либо был опубликован.⁹

ПЕССИМИЗМ ПРОТИВ ОПТИМИЗМА. Харел считает статью «СПН» и статью «Программные аспекты стратегических оборонных систем» (*Software Aspects of Strategic Defense Systems*)¹⁰ Парнаса 1984 года «чересчур безрадостными». И поэтому он стремится апеллировать к обратной стороне медали, дав своей статье подзаголовок «К более светлому будущему для развития систем». Кокс, как и Харел, прочитывает статью «СПН» как пессимистичную, он говорит: «Но если посмотреть на эти же факты с новой точки зрения, то появляется более оптимистичный вывод». Оба неверно истолковали тон.

Во-первых, моя жена, коллеги и редакторы считают, что я гораздо чаще ошибаюсь в оптимистичных прогнозах, чем в пессимистичных. В конце концов, я программист по образованию, а оптимизм — это профессиональная болезнь нашего ремесла.

В статье «СПН» говорится прямо: «Когда мы смотрим на горизонт десятилетия, мы не видим никакой серебряной пули... Однако скептицизм не есть пессимизм... Королевской дороги нет, но есть дорога». Она предсказывает что инновации, которые находятся на этапе своего развития в 1986-м, если будут разработаны и использованы, все *совместно* приведут к увеличению продуктивности на порядок. По мере того как продолжается десятилетие 1986–1996 го-

дов, это предсказание выглядит скорее слишком оптимистичным, чем слишком безрадостным.

Даже если бы статья «СПН» повсеместно воспринималась как пессимистичная, что в этом плохого? Является ли утверждение Эйнштейна о том, что ничто не может двигаться быстрее скорости света, «безрадостным» или «мрачным»? Как насчет результата Геделя, что некоторые вещи не могут быть доказаны? Статья «СПН» предпринимает попытку установить, что «сама природа программного обеспечения делает маловероятным, что когда-либо будут какие-либо серебряные пули». Турский (Turski) в своем превосходном ответном докладе на конференции IFIP красноречиво высказался:

Из всех заплутавших научных начинаний нет более жалкого, чем поиски философского камня, вещества, которое, как предполагается, превращает неблагоприятные металлы в золото. Высшая цель алхимии, которой настойчиво занимались поколения исследователей, щедро финансируемых светскими и духовными правителями, — это неразбавленный экстракт благонамеренного мышления из общего допущения, что вещи таковы, какими мы хотели бы их видеть. Это очень человеческая вера. Требуется много усилий, чтобы принять существование неразрешимых проблем. Желание увидеть выход несмотря ни на что, даже когда доказано, что его не существует, очень и очень велико. И большинство из нас испытывает большое сочувствие к этим мужественным душам, которые пытаются достичь невозможного. И так оно продолжается. Пишутся диссертации по квадратуре круга. Лосьоны для восстановления потерянных волос состряпаны и пользуются спросом. Методы повышения продуктивности программного обеспечения проклевываются и очень хорошо продаются.

Слишком часто мы склонны следовать собственному оптимизму (или эксплуатировать оптимистические надежды наших спонсоров). Слишком часто мы готовы пренебречь голосом разума и прислушаться к пению сирен толкачей панацеи.¹¹

Мы с Турским настаиваем на том, что пустые мечты *тормозят продвижение вперед и растрачивают усилия*.

«МРАЧНЫЕ» ТЕМЫ. По восприятиям Харела мрачность в статье «СПН» возникает из трех тем:

- Четкое разделение на существенные и случайные признаки.
- Изолированное рассмотрение каждого кандидата в «серебряные пули».
- Предсказание только на десять лет, а не на достаточно долгий срок, в течение которого «можно ожидать каких-либо значительных улучшений».

Что касается первого, то в этом весь смысл статьи. Я по-прежнему считаю, что это разделение является центральным для понимания того, почему программное обеспечение является трудным. Это верное руководство в том, какие виды атак предпринимать.

Что касается изолированного рассмотрения кандидатов в «серебряные пули», то это правда. Один за другим были предложены различные кандидаты, с экстравагантными требованиями для *каждого из них*. Справедливо оценивать их достоинство по одному. Это не те методы, которым я противостою, это ожидание того, что они будут работать волшебным образом. Гласс (Glass), Вессеи (Vessey) и Конджер (Conger) в своей статье 1992 года предлагают достаточно доказательств того, что тщетные поиски серебряной пули еще не закончились.¹²

Что касается выбора в качестве периода предсказания десяти лет, а не сорока, то более короткий период был отчасти уступкой тому, что наши предсказательные способности никогда не были высокими по прошествии десятилетия. Кто из нас в 1975 году предсказал микрокомпьютерную революцию 1980-х годов?

Есть и другие причины для десятилетнего лимита: все заявки, сделанные для пуль-кандидатов, имели в отношении них опре-

деленную непосредственность. Я не помню ни одного, кто сказал бы: «инвестируйте в мою панацею, и через десять лет вы начнете получать дивиденды». Более того, соотношение цена/производительность аппаратного обеспечения за десятилетие улучшилось, возможно, в 100 раз, и сравнение, хотя и совершенно неверное, подсознательно будет неизбежным. Мы, несомненно, добьемся существенного прогресса в течение следующих сорока лет; повышение на порядок за сорок лет вряд ли можно назвать волшебным.

МЫСЛЕННЫЙ ЭКСПЕРИМЕНТ ХАРЕЛА. Харел предлагает мысленный эксперимент, в котором предполагает, что статья «СПН» была написана не в 1986 году, а в 1952-м, но утверждает те же самые тезисы. Доведя это до абсурда (*reducto ad absurdum*), он использует его как аргумент против попыток отделить существенные признаки от частных.

Но этот аргумент не работает. Во-первых, статья «СПН» начинается с утверждения, что в программировании 1950-х годов частные трудности чрезвычайно доминировали над существенными, что они больше этого не делают и что их устранение привело к улучшению на порядок. Переводить этот аргумент назад на сорок лет было бы неразумно; едва ли можно представить, что в 1952 году кто-то заявлял бы, что случайные трудности не составляют главенствующей части усилий.

Во-вторых, положение дел, которое, по мнению Харела, преобладало в 1950-х годах, не соответствует действительности:

Это было время, когда вместо того, чтобы разрабатывать большие сложные системы, программисты писали обычные однопользовательские программы, которые на современных языках программирования заняли бы 100–200 строк и которые должны были выполнять скромные алгоритмические задачи. С учетом имеющейся тогда технологии и методологии такие задачи были столь же труднопреодолимы. Неудачи, ошибки и сорванные сроки были повсюду.

Затем он описывает, как постулируемые неудачи, ошибки и сорванные сроки в обыкновенных малых программах для одного человека были улучшены на порядок в течение следующих 25 лет.

Но передовые решения в 1950-х годах на самом деле не были представлены малыми программами одного человека. В 1952 году *Univac* обрабатывала данные переписи населения 1950 года с помощью сложной программы, разработанной примерно восемью программистами.¹³ Другие машины занимались химической динамикой, расчетами диффузии нейтронов, расчетами производительности ракет и т. д.¹⁴ Ассемблеры, перемещающие компоновщики и загрузчики, интерпретирующие системы с плавающей точкой и др., использовались рутинным образом.¹⁵

К 1955 году люди создавали бизнес-программы, измерявшиеся от 50 до 100 человеко-лет.¹⁶ К 1956 году *General Electric* ввела в действие систему начисления заработной платы на своем заводе в Луисвилле, включавшую более чем 80 000 слов программы. К 1957 году компьютер противовоздушной обороны SAGE ANFSQ/7 работал уже два года, и на 30 объектах действовала система связи, состоящая из 75 000 инструкций, основанных на отказоустойчивых реально-временных дуплексах.¹⁷ Вряд ли можно утверждать, что с 1952 года именно эволюция методики разработки программ одним человеком главным образом описывает усилия в инженерии программного обеспечения.

И ВОТ ОНА. Харел продолжает, предлагая свою собственную серебряную пулю, методику моделирования под названием «Ванильный фреймворк» (*The Vanilla Framework*). Сам подход не описан достаточно подробно для того, чтобы его оценить, но все же дается ссылка на статью и на технический отчет, который должен появиться в виде книги в свое время.¹⁸ Моделирование действительно затрагивает существенный признак, надлежащее изготовление и отладку концепций, поэтому вполне возможно, что «Ванильный фреймворк» будет революционным. Я надеюсь, что это так. Кен

Брукс (Ken Brooks) сообщает, что нашел эту методику полезной, когда попробовал ее на реальной работе.

НЕВИДИМОСТЬ. Харел решительно утверждает, что большая часть концептуального конструктора программного обеспечения, в сущности, является топологической по своей природе, и ее связи имеют естественные аналоги в пространственных/графических представлениях:

Использование подходящего визуального формализма может оказать заметное воздействие на инженеров и программистов. Более того, этот эффект не ограничен простыми частными вопросами; было обнаружено, что качество и оперативность самого их мышления улучшились. Успешное развитие системы в будущем будет вращаться вокруг визуальных представлений. Сначала мы будем заниматься концептуализацией, используя «надлежащие» единицы и связи. Затем сформулируем и переформулируем наши концепции как серии все более всеобъемлющих моделей, представленных в надлежащей комбинации визуальных языков. И это должна быть комбинация, поскольку системные модели имеют несколько аспектов, каждый из которых вызывает различные виды мысленных образов.

...Некоторые аспекты процесса моделирования с меньшей очевидностью, по сравнению с другими аспектами, позволяют хорошую визуализацию. Алгоритмические операции с переменными и структурами данных, например, вероятно, останутся текстовыми.

Наши с Харелом позиции весьма близки. Я утверждал, что структура программного обеспечения не встроена в трехмерное пространство, поэтому нет естественного единого отображения из концептуального дизайна в диаграмму, будь то в двух измерениях или более. Он признает, и я с ним согласен, что нужны многочисленные диаграммы, каждая из которых охватывает какой-то отдельный аспект, и что некоторые аспекты вообще не очень хорошо изображаются на диаграмме.

Полностью разделяю его энтузиазм по поводу использования диаграмм в качестве мыслительных и дизайнерских помощников. Мне давно нравится задавать кандидатам в программисты вопрос «где следующий ноябрь?». Если этот вопрос является слишком загадочным, тогда я предлагаю: «Расскажите мне о своей ментальной модели календаря». По-настоящему хорошие программисты обладают сильным пространственным чувством, у них обычно есть геометрические модели времени, и они довольно часто понимают первый вопрос без детализации. У них очень индивидуалистические модели.

ТОЧКА ЗРЕНИЯ ДЖОНСА — ПРОДУКТИВНОСТЬ СЛЕДУЕТ ЗА КАЧЕСТВОМ

Кэйперс Джонс (Capers Jones), написавший сначала серию мемуардумов, а затем книгу, предлагает проницательное понимание, которое было заявлено несколькими моими респондентами. Статья «СПН», как и большинство работ того времени, была сосредоточена на *продуктивности* — выходе программной продукции на единицу входных затрат. Джонс говорит: «Нет. Сосредоточьтесь на *качестве*, и продуктивность последует».¹⁹ Он утверждает, что дорогостоящие и опоздавшие проекты инвестируют большую часть дополнительной работы и времени в отыскание и исправление ошибок в спецификации, дизайне, имплементации. Он предлагает данные, которые показывают сильную корреляцию между нехваткой систематического контроля качества и срывами графика. Я верю в это. Бём (Boehm) указывает на то, что продуктивность снова падает, когда преследуют предельное качество, как в программном обеспечении IBM для космического шаттла.

Аналогичным образом Коки (Coqui) утверждает, что принципы систематической разработки программного обеспечения яви-

лись ответом на озабоченность не столько производительностью, сколько качеством (в особенности стремлением избежать крупных катастроф).

Но обратите внимание: целью применения принципов инженерии к производству программного обеспечения в 1970-х годах было повышение качества, тестируемости, стабильности и предсказуемости программных продуктов — но не обязательно эффективности производства программного обеспечения.

Движущей силой использования принципов инженерии программного обеспечения в производстве программного обеспечения был страх перед крупными авариями, которые могли быть вызваны неконтролируемыми творцами, ответственными за разработку все более сложных систем.²⁰

ТАК ЧТО ЖЕ СЛУЧИЛОСЬ С ПРОДУКТИВНОСТЬЮ?

ПОКАЗАТЕЛИ ПРОДУКТИВНОСТИ. Показатели продуктивности очень трудно определить, откалибровать и отыскать. Каперс Джонс считает, что для двух эквивалентных программ на Cobol, написанных с разницей в 10 лет, одна без структурной методологии и одна с ней, выигрыш является трехкратным.

Эд Йордон (Ed Yourdon) говорит:

Я вижу, что возможно получить пятикратное улучшение за счет рабочих станций и программных инструментов». Том Демарко (Tom De Marco) считает, что «ваше ожидание улучшения на порядок за 10 лет благодаря целой корзине методик было оптимистичным. Я не видел организаций, которые бы улучшили ситуацию на порядок.

КОММЕРЧЕСКОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ — ПОКУПАЙТЕ, А НЕ РАЗРАБАТЫВАЙТЕ. Одна из оценок 1986 года в статье «СПН», как мне кажется,

оказалась верной: «Развитие массового рынка является... самым глубоким долгосрочным трендом в инженерии программного обеспечения». С точки зрения дисциплины массовое программное обеспечение — это почти новая отрасль по сравнению с разработкой заказного программного обеспечения, будь то внутреннее или внешнее. Когда программы продаются миллионами — пусть даже тысячами — доминирующими вопросами становятся качество, своевременность, а также производительность продукта и стоимость поддержки, а не стоимость разработки, которая так важна для заказных систем.

«СИЛОВЫЕ» ИНСТРУМЕНТЫ ДЛЯ УМА. Самый впечатляющий способ повысить продуктивность программистов и управленческих информационных систем (MIS, или информационных систем управления) — это пойти в местный компьютерный магазин и купить с полки то, что они построили бы. Это не смешно; наличие дешевого, мощного коммерческого программного обеспечения удовлетворило многие потребности, которые ранее служили бы поводом для заказных пакетов. Эти «силовые» инструменты для ума больше похожи на электрические дрели, пилы и шлифовальные станки, чем на большие сложные производственные инструменты. Интеграция их в совместимые и взаимосвязанные комплекты, такие как Microsoft Works и более интегрированные Claris Works, дает огромную гибкость. И подобно набору ручных электроинструментов, частое использование небольшого набора для самых разнообразных работ делает их использование привычным. Такие инструменты должны подчеркивать простоту использования для обычного пользователя, а не для профессионала.

Айвен Селин (Ivan Selin), председатель правления American Management Systems, Inc., в 1987 году мне написал следующее:

Я придираюсь к вашему утверждению, что пакеты на самом деле не так сильно изменились... Думаю, вы слишком легко отбрасываете первостепенные последствия

вашего наблюдения, что (программные пакеты) «могут быть несколько более обобщенными и несколько более настраиваемыми под клиента, чем раньше, но не намного». Даже принимая это утверждение за чистую монету, я считаю, что пользователи видят пакеты как более обобщенные и более простые в настройке под свои нужды и что это восприятие приводит к тому, что пользователи предпочитают пакеты. В большинстве случаев, которые обнаруживает моя компания, именно (конечные) пользователи, а не люди со стороны программного обеспечения неохотно используют пакеты, опасаясь, что они потеряют существенные свойства или функции, и поэтому перспектива легкой настройки под свои нужды является для них весомым коммерческим аргументом.

Думаю, что Селин совершенно прав, — я недооценил как степень настраиваемости пакета под конкретные нужды, так и его важность.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ — ГОДИТСЯ ЛИ МЕДНАЯ ПУЛЯ?

СБОРКА БОЛЬШИМИ ЧАСТЯМИ. Иллюстрация, открывающая эту главу, напоминает нам, что, если собирать вместе множество частей, каждая из которых может быть сложной сама по себе и все из которых по дизайну имеют единообразные интерфейсы, то быстро соберутся довольно насыщенные структуры.

Один из взглядов на объектно-ориентированное программирование состоит в том, что оно является дисциплиной, которая обеспечивает *модульность* и чистые интерфейсы. Другой взгляд акцентируется на *инкапсуляции*, том факте, что невозможно увидеть внутреннюю структуру частей, а тем более дизайн. Еще один взгляд подчеркивает *наследование*, с его сопутствующей *иерархической* структурой классов, с виртуальными функциями. И еще один взгляд базируется на *сильной абстрактной типизации данных* с ее заверением

в том, что конкретный тип данных будет управляться только соответствующими ему операциями.

Теперь любая из этих дисциплин может быть получена без использования всего пакета Smalltalk или C++ — многие из них предшествовали объектно-ориентированной технологии. Привлекательность объектно-ориентированного подхода та же, что и у поливитаминной таблетки: одним махом (то есть переобучением программиста) получаешь всё. Это очень перспективная концепция.

ПОЧЕМУ ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МЕТОДИКА РАЗВИВАЕТСЯ МЕДЛЕННО?

За 9 лет, прошедших после публикации статьи «СПН», ожидания неуклонно росли. Почему рост был медленным? Теорий предостаточно. Джеймс Коггинс (James Coggins), автор четырехлетней колонки *The Best of comp.lang.c++*, в статье «Отчет о C++» (*The C++ Report*) предлагает следующее объяснение:

Проблема в том, что программисты в 0-0 практиковали применение кровосмешения и стремились к низкому в абстракциях, а не к высокому. Например, они строили такие классы, как `связный_список` или `множество`, вместо таких классов, как `пользовательский_интерфейс` или `пучок_излучения_радиации` или `модель_конечных_элементов`. К сожалению, та самая строгая проверка типов в C++, которая помогает программистам избегать ошибок, также затрудняет создание больших вещей из маленьких.²¹

Он возвращается к базовой задаче сборки программного обеспечения и утверждает, что одним из способов удовлетворения неудовлетворенных потребностей в программном обеспечении является увеличение численности интеллектуального персонала за счет задействования наших клиентов и сотрудничества с ними. В следующей ниже цитате приводятся доводы в пользу нисходящего дизайна:

Если мы проектируем крупные классы, представляющие концепции, с которыми наши клиенты уже работают, то они в состоянии понимать и критиковать проект

по мере его развития, а также вместе с нами разрабатывать контрольные примеры. Офтальмологов, с которыми я работаю, не волнует организация стека; их волнует описание формы роговицы с помощью полиномов Лежандра. Маленькая инкапсуляция дает и маленькую выгоду.

Дэвид Парнас, чья статья была одним из истоков объектно-ориентированных концепций, видит этот вопрос по-другому. Он пишет мне следующее:

Ответ прост. Все потому, что [0-0] было привязано к разнообразию сложных языков. Вместо того чтобы учить людей тому, что 0-0 — это один из подходов дизайна, и дать им принципы такого дизайна, людей учили тому, что 0-0 является способом использования отдельно взятого инструмента. Мы можем писать хорошие или плохие программы с помощью любого инструмента. Если мы не научим людей заниматься разработкой, то языки будут мало что значить. В результате имеем то, что с помощью этих языков люди создают плохие дизайны и получают от них очень мало пользы. Если польза мала, то она не приживется.

ЗАВЫШЕННЫЕ ПЕРВОНАЧАЛЬНЫЕ ЗАТРАТЫ, ВЫГОДЫ ВПОСЛЕДСТВИИ. Я лично убежден, что объектно-ориентированная методика — это особенно тяжелый случай болезни, которая характеризует многие методологические усовершенствования. Первоначальные затраты являются очень значительными — к ним относится прежде всего переобучение программистов думать совершенно по-новому, но также и дополнительные вложения в придание функциям формы обобщенных классов. Выгоды, которые я считаю реальными, а не мнимыми, возникают на протяжении всего цикла разработки; но большие выгоды окупаются в ходе последующих работ по созданию, расширению и сопровождению. Коггинс (Coggins) говорит: «Объектно-ориентированная методика не ускорит разработку ни первого проекта, ни следующего. Пятый в этом семействе пройдет молниеносно».²²

Делать реальный вклад сейчас, в надежде на предполагаемую, но не гарантированную прибыль в будущем, — это то, чем инвесторы занимаются каждый день. Однако во многих организациях, занимающихся программированием, для таких действий требуется подлинная менеджерская смелость, а это гораздо более редкий товар, чем техническая компетентность или административное мастерство. Я считаю, что крайняя степень завышенных первоначальных затрат и ориентированность на получение выгод впоследствии является самым крупным и единственным фактором, замедляющим внедрение методики О-О. Тем не менее C++, по-видимому, неуклонно заменяет C во многих сообществах.

ЧТО НАСЧЕТ ПЕРЕИСПОЛЬЗОВАНИЯ?

Лучший способ избавиться от существенного признака разработки программного обеспечения — это не создавать его вообще. Пакетное программное обеспечение является лишь одним из способов это сделать. Повторное использование программы — другой способ. Действительно, обещание легкого переиспользования классов, с легкой настройкой под конкретные нужды через наследование, является одной из самых сильных привлекательных черт объектно-ориентированной методики.

Как это часто бывает, когда человек получает некоторый опыт работы с новым способом ведения бизнеса, новый режим оказывается не таким простым, как это представлялось на первый взгляд.

Разумеется, программисты всегда использовали свою собственную работу повторно. Джонс (Jones) пишет следующее:

Большинство опытных программистов имеют персональные библиотеки, позволяющие им разрабатывать программное обеспечение, в котором примерно 30% кода от общего объема используется повторно. Возможность многократного

использования на корпоративном уровне нацелена на 75% повторно используемого кода в суммарном объеме и требует специальной библиотечной и административной поддержки. Корпоративный код многоразового использования также влечет за собой изменения в практиках учета и измерения проектов, отдавая должное возможности многоразового использования.²³

У. Хуан (W. Huang) предложил организовать фабрики программного обеспечения с матричным менеджментом функциональных специалистов, с тем чтобы воспользоваться естественной склонностью каждого к повторному использованию своего собственного кода.²⁴

Ван Шнайдер (Van Snyder) из JPL обратил мое внимание на то, что сообщество математического программного обеспечения имеет давнюю традицию переиспользования программного обеспечения:

Мы предполагаем, что барьеры для повторного использования находятся не на стороне производителя, а на стороне потребителя. Если инженер программного обеспечения, потенциальный потребитель стандартизированных программных компонентов, ощущает, что будет дороже отыскать компонент, удовлетворяющий его потребности, и проверить, что он их действительно удовлетворяет, чем написать его заново, то будет написан новый, дублирующий компонент. Обратите внимание, что выше мы сказали «по ощущениям». Не имеет значения, какой является истинная стоимость реконструкции.

Многоразовое использование было успешным для математического программного обеспечения по двум причинам: (1) оно является трудным для понимания, требующим огромного интеллектуального вклада на строку кода; и (2) существует насыщенная и стандартная номенклатура, а именно математика, для описания функциональности каждого компонента. Таким образом, стоимость реконструкции компонента математического программного обеспечения является высокой, а стоимость обнаружения функциональности существующего компонента — низкой. У профессиональных журналов есть давняя традиция публиковать и со-

бирать алгоритмы, предлагая их по умеренной цене, и коммерческих концернов, предлагающих алгоритмы очень высокого качества по несколько более высокой, но все же скромной цене. Это делает обнаружение компонента, отвечающего потребностям, проще, чем во многих других дисциплинах, где иногда невозможно точно и кратко определить свои потребности. Эти факторы сотрудничают, делая привлекательным не изобретение математического программного обеспечения, а его переиспользование.

Тот же феномен переиспользования встречается по тем же причинам в ряде сообществ, которые, к примеру, строят коды для ядерных реакторов, климатических и океанических моделей. Каждое такое сообщество выросло на одинаковых учебниках и стандартных нотациях.

Как сегодня обстоят дела с переиспользованием на корпоративном уровне? Много исследований, относительно мало практики в США, недостоверные сообщения о большем переиспользовании за рубежом.²⁵

Джонс сообщает, что все клиенты его фирмы с более чем 5000 программистами имеют формальные исследования в области переиспользования, в то время как менее 10% клиентов с менее чем 500 программистами занимаются тем же.²⁶ Он сообщает, что в индустриях с наибольшим потенциалом переиспользования исследования по возможности многократного использования (не развертывания) «являются активными и энергичными, хотя еще и не совсем успешными». Эд Йордон сообщает о программной фирме в Маниле, в которой пятьдесят из двухсот программистов создают только многократные модули для использования остальными. «Я видел несколько случаев — принятие связано не с техническими, а с *организационными* факторами, такими как структура премирования».

Демарко сообщает мне, что доступность пакетов массового рынка и их пригодность в качестве поставщиков обобщенных функций,

например системы баз данных, существенно снизили как давление, так и предельную полезность многоразового использования модулей собственного прикладного кода. «Многоразовые модули, как правило, так или иначе являлись обобщенными функциями».

Парнас пишет следующее:

Повторное использование — гораздо легче сказать, чем сделать. Для этого требуется как хороший дизайн, так и очень хорошая документация. Даже когда мы видим отличный дизайн, что по-прежнему бывает нечасто, мы не видим, что компоненты используются повторно без хорошей документации.

Кен Брукс комментирует трудность предвосхищения того, какое обобщение окажется необходимым: «Мне все время приходится выкручиваться даже при пятом использовании моей личной библиотеки пользовательского интерфейса».

Реальное переиспользование, похоже, только начинается. Джонс утверждает, что несколько переиспользуемых модулей кода предлагаются на открытом рынке по ценам от 1 до 20 % от обычных затрат на разработку.²⁷ Демарко говорит следующее:

Меня обескураживает весь этот феномен повторного использования. Для повторного использования почти полностью отсутствует теорема его существования. Время подтвердило, что в изготовлении переиспользуемых вещей существуют большие издержки.

Йордон дает оценку больших издержек: «Хорошее эмпирическое правило в том, что такие переиспользуемые компоненты потребуют в два раза больше усилий, чем “одноразовый компонент”».²⁸ Я вижу эти издержки именно как усилия по коммерческому внедрению компонента, рассмотренные в главе 1. Так что моя оценка соотношения усилий будет трехкратной.

Ясно, что мы видим много форм и разновидностей многоразового использования, но не столько, как мы ожидали к настоящему времени. Нам еще многое предстоит усвоить.

УСВОЕНИЕ БОЛЬШИХ СПИСКОВ КОМАНД — ПРЕДСКАЗУЕМАЯ, НО НЕ ПРЕДСКАЗАННАЯ ПРОБЛЕМА ДЛЯ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ ПО

Чем выше уровень мышления, на котором думаешь, тем многочисленнее примитивные мыслительные элементы, с которыми приходится иметь дело. Поэтому языки программирования являются гораздо более сложными, чем машинные языки, а естественные языки еще сложнее. Языки более высокого уровня имеют более обширный список команд, более сложный синтаксис и более насыщенную семантику.

Как научная дисциплина, мы не взвесили последствия этого факта для повторного использования программ. Чтобы повысить качество и производительность, мы хотим строить программы из частей с отлаженными функциями, которые существенно выше по уровню, чем операторы языков программирования. Поэтому, делаем ли мы это с помощью библиотек классов или библиотек процедур, нам придется столкнуться с резким увеличением размеров наших словарей программирования. Изучение списков команд составляет значительную часть препятствий к повторному использованию.

Поэтому сегодня у людей есть библиотеки классов более чем с 3000 членами. Многие объекты требуют конкретизации от 10 до 20 параметров и настроечных переменных. Любой программист, работающий с этой библиотекой, должен усвоить синтаксис (внешние

интерфейсы) и семантику (детальное функциональное поведение) ее членов, если он хочет достичь всего потенциала повторного использования.

Эта работа далеко не безнадежна. Носители языка обычно используют вокабуляр из более чем 10 000 слов, образованные люди гораздо больше. Каким-то образом мы усваиваем синтаксис и едва уловимую семантику. Мы правильно различаем смысл слов «гигантский», «огромный», «необъятный», «непомерный», и люди не говорят о «непомерных пустынях» или «необъятных элементах».

Нам нужны исследования, которые приспособят большой объем языкознания для задачи многоразового использования программного обеспечения. Некоторые из уроков очевидны:

- Люди учатся в контексте предложений, поэтому нам нужно публиковать много примеров скомпонованных продуктов, а не только библиотеки частей.
- Люди не запоминают ничего, кроме орфографии. Они усваивают синтаксис и семантику постепенно, в контексте, используя язык.
- Люди группируют правила композиции слов по синтаксическим классам, а не по совместимым подмножествам объектов.

ЧИСТЫЙ ИТОГ ПО ПУЛЯМ — ПОЛОЖЕНИЕ НЕ ИЗМЕНИЛОСЬ

Итак, мы возвращаемся к основам. Сложность — *это то*, чем мы занимаемся, и сложность — *это то*, что нас ограничивает. Р. Л. Гласс (R. L. Glass) в своей статье 1988 года точно резюмирует мои взгляды 1995 года:

Так что же, оглядываясь назад, сказали нам Парнас и Брукс? Что развитие программного обеспечения является концептуально трудным бизнесом. Что волшебные решения не ждут нас за следующим углом. Что пришло время для практиков протестировать эволюционные улучшения и не ждать — или надеяться — на революционные.

Некоторые специалистов области инженерии ПО считают, что эта картина является обескураживающей. Это те, кто все еще думал, что прорывы где-то рядом.

Но некоторые из нас — те, кто достаточно упрям, чтобы считать себя реалистами, — воспринимают это как глоток свежего воздуха. Наконец-то мы можем сосредоточиться на чем-то более жизнеспособном, чем журавль в небе. Теперь, вероятно, мы сможем приступить к постепенным улучшениям продуктивности ПО, которые возможны, вместо того чтобы ждать прорывов, которые вряд ли когда-либо произойдут.²⁹



Брукс излагает свои тезисы. Источник: Википедия

18

ТЕЗИСЫ МИФИЧЕСКОГО ЧЕЛОВЕКО-МЕСЯЦА: FALSE ИЛИ TRUE?

Ведь краткость — лучший выбор тут,
Поймут нас или не поймут.

Сэмюэл Батлер. «Гудибра»

Сегодня об инженерии программного обеспечения известно гораздо больше, чем было известно в 1975 году. Какие утверждения в оригинальном издании 1975 года были подтверждены данными и опытом? Какие были опровергнуты? Какие из них потеряли актуальность в связи с изменениями, произошедшими в мире? Чтобы вам легче было судить, здесь, в виде сводки, приведены важнейшие утверждения книги 1975 года, которые я считал верными: факты и извлеченные из опыта практические правила, приведенные с сохранением изначального смысла. (Вы можете спросить: «Если это все, что сказано в оригинальной книге, то почему потребовалось 177 страниц^{*}, чтобы это высказать?») Комментарии в скобках являются новыми.

Большинство этих тезисов можно проверить на практике. Надеюсь, что, изложив их в структурированном виде, я смогу упорядочить мысли, измерения и комментарии читателей.

ГЛАВА 1. СМОЛЯНАЯ ЯМА

1.1. Создание систем программирования как продукта требует усилий примерно в 9 раз больше, чем компонентные программы, написанные отдельно для частного использования. По моей оценке, коммерческое внедрение программного продукта увеличивает стоимость втрое; проектирование, интеграция и тестирование компонентов, которые должны образовать согласованную систему, также вводит коэффициент, равный трем; и эти стоимостные компоненты, по существу, не зависят друг от друга.

1.2. Ремесло программирования «удовлетворяет творческие устремления, заложенные глубоко внутри нас, и восхищает чувства,

^{*} Имеется в виду объем оригинала на английском языке. — *Примеч. ред.*

которые роднят нас со всеми людьми», обеспечивая пять видов радостей:

- Радость от созидания вещей.
- Радость от созидания вещей, полезных другим людям.
- Очарование от конструирования головоломных объектов, состоящих из взаимосвязанных движущихся частей.
- Радость от постоянного усвоения нового — этого предостаточно в неповторяющейся работе.
- Наслаждение от работы в среде, столь податливой, — чистой мысленной материи, которая, тем не менее, существует, движется и действует отличным от словесных объектов образом.

1.3. Схожим образом данному ремеслу присущи особые горести:

- Необходимость приспособливаться к требованиям совершенства — самая трудная часть освоения программирования.
- Постановка задач осуществляется другими людьми, и приходится зависеть от вещей (особенно программ), которые нельзя контролировать; полномочия не соответствуют ответственности.
- Это звучит хуже, чем есть на самом деле: фактические полномочия приходят по мере достижения результатов.
- С любым творчеством приходят монотонные часы напряженного труда. Программирование не является исключением.
- Вопреки ожиданиям, что программный проект будет сходиться быстрее по мере приближения к финальной стадии, на практике чем ближе к концу, тем медленнее идет работа.
- Всегда существует опасность того, что продукт устареет еще до его завершения. Живой тигр никогда не сравнится с бумажным, только если не потребуется его реальное использование.

ГЛАВА 2. МИФИЧЕСКИЙ ЧЕЛОВЕКО-МЕСЯЦ

2.1. Из-за нехватки календарного времени провалилось программных проектов больше, чем по всем другим причинам, вместе взятым.

2.2. Высокая кухня требует времени; некоторые работы нельзя торопить, не испортив результат.

2.3. Программисты, как один, являются оптимистами: «все будет хорошо».

2.4. Поскольку программист при создании продукта имеет дело с чистым мысленным материалом, мы не ждем особых трудностей в имплементации.

2.5. Но сами наши *идеи* бывают ошибочными — отсюда и ошибки в программах.

2.6. Наша методика оценивания, построенная на учете затрат, заставляет путать приложенные усилия и продвижение вперед. *Человеко-месяц является ошибочным и опасным мифом, поскольку из него следует, что люди и месяцы — взаимозаменяемы.*

2.7. Разделение работы между несколькими людьми влечет за собой необходимость в дополнительных усилиях на обучение и коммуникацию.

2.8. Мое эмпирическое правило следующее: $\frac{1}{3}$ времени на разработку, $\frac{1}{6}$ на написание кода, $\frac{1}{4}$ на тестирование компонентов и $\frac{1}{4}$ на тестирование системы.

2.9. Как дисциплине (в смысле профессии), нам не хватает данных для того, чтобы делать оценки.

2.10. Поскольку мы не уверены в своих оценках относительно сроков работ, нам часто не хватает смелости каждый раз защищать их от давления со стороны менеджмента и клиентов.

2.11. Закон Брукса: добавление рабочей силы (людей/разработчиков) в запаздывающий программный проект задержит его еще больше.

2.12. Добавление людей в проект по созданию программного обеспечения требует увеличения суммарного усилия по трем направлениям: [необходимость] перераспределения работы и вызванное этим нарушение хода текущих работ, [необходимость] обучения новых людей, и увеличение [накладных] расходов на коммуникацию [в большем коллективе].

ГЛАВА 3. ХИРУРГИЧЕСКАЯ БРИГАДА

3.1. Очень хорошие профессиональные программисты в *10 раз* продуктивнее слабых, при наличии того же уровня обучения и двух-летнего опыта. (Сакман, Грант и Эриксон)

3.2. Данные Сакмана, Гранта и Эриксона не показали никакой корреляции между опытом и производительностью. Я сомневаюсь в универсальности этого результата.

3.3. Лучше всего небольшая высококлассная команда — чем меньше умов, тем лучше.

3.4. Часто лучше всего, если команда состоит из двух человек, один из которых является лидером. [Вспомните про Божий план супружества.]

3.5. Небольшая высококлассная команда является слишком медленной для действительно крупных систем.

3.6. Большинство примеров опыта с действительно крупными системами показывают, что подход к вертикальному масштабированию на основе грубой силы является дорогостоящим, медленным,

неэффективным и выдает системы, которые не являются концептуально целостными.

3.7. Организация по типу хирургических бригад с главным программистом позволяет достичь целостности продукта благодаря его проектированию в нескольких головах и общей продуктивности благодаря наличию многочисленных помощников при радикально сокращенном обмене информацией.

ГЛАВА 4. АРИСТОКРАТИЯ, ДЕМОКРАТИЯ И СИСТЕМНЫЙ ДИЗАЙН

4.1. «Концептуальная целостность является *наиболее важным* соображением в системном дизайне».

4.2. «*Отношение* функциональности к концептуальной сложности является конечной проверкой дизайна системы», а не только уровень насыщенности функциональностью. [Это отношение является мерой простоты использования, справедливой как для простого, так и для продвинутого использования.]

4.3. Для достижения концептуальной целостности дизайн должен исходить из одного ума или небольшой группы согласующихся умов.

4.4. «Отделение архитектуры от имплементации является мощным способом получения концептуальной целостности на очень крупных проектах». [И на маленьких тоже.]

4.5. «Если система должна обладать концептуальной целостностью, то кто-то должен контролировать стоящие за ней концепции. И это будет аристократия, которая не обязана ни перед кем извиняться».

4.6. Дисциплина совершенствует мастерство. Архитектура, представленная извне, усиливает, а не сводит к нулю творческие возможности имплементирующей группы.

4.7. Концептуально целостная система разрабатывается и тестируется быстрее.

4.8. Зачастую процессы создания архитектуры, программная имплементация и аппаратная реализация программного обеспечения могут проходить параллельно. [Параллельно можно осуществлять дизайн аппаратного и программного обеспечения.]

ГЛАВА 5. ЭФФЕКТ ВТОРОЙ СИСТЕМЫ

5.1. Связь, установленная на ранних этапах и продолжающаяся непрерывно, может дать архитектору верную оценку стоимости, а разработчику — уверенность в проекте, не снимая при этом четкого разграничения зон ответственности.

5.2. Как архитектор может успешно влиять на имплементацию:

- Помнить, что творческую ответственность за имплементацию несет строитель; архитектор только предлагает идеи.
- Быть готовым предложить способ имплементации всего, что он [архитектор] предлагает; быть готовым принять любой другой столь же хороший способ.
- Разбираться спокойно и конфиденциально в таких предложениях.
- Быть готовым отказаться от признания заслуг за предложенные улучшения.
- Прислушиваться к предложениям разработчика по улучшению архитектуры.

5.3. Вторая система является самой опасной для человека, который ее проектирует; общая тенденция — делать дизайн с большим запасом.

5.4. OS/360 является хорошим примером эффекта второй системы. [Windows NT, похоже, является примером 1990-х.]

5.5. Полезной является дисциплина задания *априорных* ограничений для функции в байтах и микросекундах [заранее].

ГЛАВА 6. ДОВЕДЕНИЕ ДО СВЕДЕНИЯ

6.1. Даже когда команда разработчиков велика, результаты должны быть сведены к написанному одним или двумя ее членами ради непротиворечивости мини-решений.

6.2. Важно четко определить те части архитектуры, которые *не* прописаны так же тщательно, как те, которые прописаны.

6.3. Требуется как формальное определение дизайна, для точности, так и неформальное определение, для понятности.

6.4. Одно из формальных и текстовых определений должно быть стандартным, а другое производным. Любое определение может служить в любой роли.

6.5. Хотя имплементация, включая моделирование, может служить архитектурным определением, однако такой подход имеет существенные недостатки.

6.6. Прямое встраивание является очень чистым методом для обеспечения соблюдения архитектурных стандартов в программном обеспечении [В аппаратном обеспечении тоже. К примеру, интерфейс Mac WIMP встроен в ROM].

6.7. Архитектурное «определение станет чище и [архитектурная] дисциплина жестче, если первоначально построены хотя бы две имплементации».

6.8. Важно разрешить архитекторам давать быстрые ответы на частые вопросы имплементаторов; необходимо регистрировать их и публиковать.

6.9. «Лучший друг менеджера проекта — его ежедневный противник, независимая организация по тестированию продукта».

ГЛАВА 7. ПОЧЕМУ ПРОВАЛИЛСЯ ВАВИЛОНСКИЙ ПРОЕКТ

7.1. Проект Вавилонской башни потерпел крах из-за отсутствия *коммуникации* и невозможности ее *организовать* впоследствии.

КОММУНИКАЦИЯ

7.2. «Срыв проекта, функциональная несогласованность и системные ошибки возникают потому, что левая рука не знает, что делает правая». Команды расходятся в принимаемых допущениях.

7.3. Команды должны обмениваться информацией друг с другом как можно более разнообразными способами: неофициально, путем проведения регулярных совещаний по проекту с техническими брифингами и через общую официальную рабочую книгу проекта [В 65-м был телефон, в 95-м e-mail]*.

* А в 2020-м — это видеоконференции и чаты. — *Примеч. ред.*

РАБОЧАЯ КНИГА ПРОЕКТА

7.4. Рабочая книга проекта — «это не столько отдельный документ, сколько структура, налагаемая на документы, которые проект так или иначе будет производить».

7.5. «Все документы проекта должны быть частью этой структуры [рабочей книги]».

7.6. Структура рабочей книги должна быть спланирована *тщательно и заблаговременно*.

7.7. Надлежащее структурирование текущей документации с самого начала «оформляет последующие документы в сегменты, которые вписываются в эту структуру», что улучшит справочное руководство по продукту.

7.8. «Каждый член команды должен видеть материал [рабочей книги] *в полном объеме*». [Я бы сейчас сказал, что каждый член команды *должен иметь возможность* видеть все это. Просмотра web-страниц было бы достаточно.]

7.9. Своевременное обновление имеет решающее значение.

7.10. Для пользователя [читателя документации] важно, чтобы изменения, внесенные в документ с момента, когда он читал его в прошлый раз, были явно выделены, а также была отмечена значимость внесенных изменений.

7.11. Рабочая книга проекта OS/360 началась с бумаги, затем ее перенесли на микрофиши.

7.12. Сегодня [даже в 1975 году] совместно используемая электронная записная книжка является гораздо более эффективным, дешевым и простым механизмом для достижения всех этих целей.

7.13. По-прежнему необходимо отмечать текст полосами изменений и датами пересмотра [или их функциональным эквивалентом]. По-прежнему необходима электронная сводка изменений с режимом доступа LIFO*.

7.14. Парнас с полной уверенностью утверждает, что цель, когда каждый видит все, является *абсолютно неверной*. Части должны быть инкапсулированы так, чтобы никому не требовалось или не позволялось видеть внутренности каких-то частей, кроме своей части работы. Видны должны быть только интерфейсы.

7.15. Предложение Парнаса ведет к катастрофе. [*Парнас, убедив меня в обратном, совершенно изменил мое мнение.*]

ОРГАНИЗАЦИЯ

7.16. Целью организации является сокращение необходимых объемов коммуникации и усилий по координации.

7.17. Организация воплощает *разделение труда* и *специализацию функций*, чтобы избежать избыточной коммуникации.

7.18. Традиционная древовидная структура организации отражает принцип *полномочий*, согласно которому ни один человек не может служить двум хозяевам.

7.19. Структура *коммуникации* в организации — это сеть, а не дерево, поэтому для преодоления коммуникативных недостатков древовидной структуры организации необходимо разработать всевозможные специальные организационные механизмы («пунктирные линии»).

* LIFO — «последним пришел — первым вышел», имеется в виду, что последние изменения должны быть видны в первую очередь. — *Примеч. ред.*

7.20. Каждый подпроект содержит две лидерские роли: роль *продюсера* и роль *технического директора*, или архитектора. Функции этих двух ролей весьма различны и требуют различных талантов.

7.21. Между этими ролями формируются три взаимосвязи, в равной степени эффективные:

- Продюсер и директор могут быть равны.
- Продюсер может быть начальником, а директор — правой рукой продюсера.
- Директор может быть начальником, а продюсер — правой рукой директора.

ГЛАВА 8. ПОПЫТКИ ИЗМЕРИТЬ

8.1. Невозможно точно оценить общее усилие или график проекта программирования, просто умножив время на написание кода на коэффициенты для других частей работы.

8.2. Данные для создания изолированных малых систем неприменимы к проектам систем программирования.

8.3. Программирование увеличивает степень размера программы.

8.4. Некоторые опубликованные исследования показывают, что экспонента составляет около 1, 5. [*Данные Бема совершенно с этим не согласуются, но варьируются от 1, 05 до 1,2.*]¹

8.5. Данные Портмана по *ICL* показывают, что штатные программисты тратят только около 50 % своего времени на программирование и отладку по сравнению с другими добавочными работами.

8.6. По данным Арона из *IBM*, производительность труда лежит в пределах от 1,5 до 10 тысяч строк кода (KLOC) на человека в год,

в зависимости от количества взаимодействий между частями системы.

8.7. Данные Харра по *Bell Labs* показывают продуктивности по работе, связанной с операционной системой и подобным обеспечением, которая выполняется около 0,6 KLOC/человеко-лет, и работе, связанной с компилятором и подобным обеспечением, которая выполняется около 2,2 KLOC/человеко-лет для готовых продуктов.

8.8. Данные Брукса по OS/360 согласуются с данными Харра: 0,6–0,8 KLOC/человеко-лет над операционными системами и 2–3 KLOC/человеко-лет над компиляторами.

8.9. Данные Корбатто по проекту *MULTICS* университета MIT показывают продуктивность 1,2 KLOC/человеко-лет при разработке смеси операционных систем и компиляторов, но это строки кода на PL/1, тогда как все остальные данные являются строками ассемблерного кода!

8.10. Продуктивность кажется постоянной с точки зрения элементарных инструкций языка.

8.11. При использовании подходящего языка программирования высокого уровня продуктивность разработки может быть увеличена в 5 раз.

ГЛАВА 9. НЕПОСИЛЬНЫЙ ГРУЗ

9.1. Не только время выполнения, затраченное на программу, но и *объем памяти* является первостепенной стоимостью. Это особенно верно для операционных систем, значительная часть которых остается резидентной.

9.2. Несмотря на это, деньги, потраченные на память для размещения программы, дают очень хорошее улучшение характеристик на

единицу вложений — лучшее, чем все другие способы инвестирования в конфигурацию. Плох не размер программы, а лишний размер.

9.3. Создатель программного обеспечения должен устанавливать размерные цели, управлять размером и разрабатывать методы уменьшения размера, как это делает строитель аппаратного обеспечения для компонентов.

9.4. Бюджеты размеров должны быть явными не только в отношении размера резидентности, но и в отношении доступа к диску, обусловливаемого выборками программ.

9.5. Бюджеты размеров должны быть привязаны к распределениям функций; следует точно определять, что именно модуль должен делать при указании требуемого размера.

9.6. В крупных командах подкоманды тяготеют к субоптимизации ради достижения своих собственных целей вместо того, чтобы думать о суммарном эффекте на пользователя. Такое неверное толкование цели является главной опасностью крупных проектов.

9.7. В ходе всей имплементации архитекторы системы должны с неусыпной бдительностью непрерывно обеспечивать целостность системы.

9.8. Наиболее важной функцией менеджера по программированию вполне может быть формирование общесистемного, ориентированного на пользователя отношения.

9.9. Раннее политическое решение — определить, насколько мелкозернистым будет выбор опций пользователем, поскольку объединение опций в группы экономит пространство памяти [и часто снижает маркетинговые затраты].

9.10. Размер переходной области, а следовательно, и объем программы в расчете на выборку из диска являются решающими

показателями, поскольку производительность считается сверхлинейной функцией этого размера. [Это требование устарело благодаря наличию виртуальной памяти, а затем дешевой реальной памяти. Пользователи теперь, как правило, покупают достаточный объем реальной памяти, который вмещает весь код главных приложений.]

9.11. Для достижения хороших компромиссов между пространством и временем команда должна быть обучена методике программирования, свойственной определенному языку или машине, в особенности если они новые.

9.12. Программирование имеет свою технологию, и каждый проект нуждается в библиотеке стандартных компонентов.

9.13. Библиотеки программ должны иметь две версии каждого компонента: быструю и сжатую. [Сегодня это выглядит устаревшим.]

9.14. Компактные и быстрые программы почти всегда являются результатом *стратегического прорыва*, а не тактического ума.

9.15. Часто таким прорывом будет являться новый алгоритм.

9.16. Чаще всего прорыв будет проистекать из переделки представления моделей данных или таблиц. *Представление — это существенный признак программирования.*

ГЛАВА 10. ДОКУМЕНТАРНАЯ ГИПОТЕЗА

10.1. «Среди потока бумаг небольшое число документов становится критически важными, вокруг них вращается управление каждым проектом. Эти документы являются ключевыми личными инструментами менеджера».

10.2. Для проекта разработки компьютера важнейшими документами являются целевые критерии, справочное руководство, сроки, бюджет, организационная схема, распределение площадей, а также оценка, прогноз и цена самой машины.

10.3. Для университетской кафедры важнейшие документы похожи: целевые критерии, требования к диплому, описания курсов, предложения по исследованиям, расписание занятий и учебный план, бюджет, распределение площадей и назначения сотрудников и аспирантов.

10.4. Для проекта по созданию программного обеспечения потребности являются одинаковыми: целевые критерии, справочное руководство пользователя, внутренняя документация, график, бюджет, организационная схема и распределение площадей.

10.5. Следовательно, менеджер обязан с самого начала оформить необходимый комплект документов, даже на малом проекте.

10.6. Подготовка каждого документа из этого небольшого комплекта фокусирует мысль и кристаллизует обсуждение. Акт написания требует сотен мини-решений, и именно их существование отличает четкие, точные политики от нечетких.

10.7. Ведение каждого критически важного документа обеспечивает механизм наблюдения за состоянием дел и предупреждения сбоев.

10.8. Каждый документ сам по себе служит контрольным списком и базой данных.

10.9. Главная задача менеджера проекта — удерживать всех в одном направлении.

10.10. Главной ежедневной работой менеджера проекта является коммуникация, а не принятие решений; документы сообщают планы и решения всей команде.

10.11. Только малую часть времени — возможно, 20 % — менеджер технического проекта тратит на выполнение работ, которые требуют сведений, отсутствующих в его памяти.

10.12. По этой причине расхваливаемая рыночная концепция «управленческой системы» для поддержки исполнительного персонала не основана на действительной модели поведения исполнителей.

ГЛАВА 11. ПЛАНИРУЙ ВЫБРОСИТЬ

11.1. Инженеры-химики научились не переносить процесс с лабораторного стенда на завод за один шаг, а предпочитают создавать *пилотную установку*, дающую опыт в вертикальном масштабировании объемов и эксплуатации в незащищенных средах.

11.2. Этот промежуточный этап в равной степени необходим и для программных продуктов, но у инженеров программного обеспечения еще не превратилось в рутину проведение полевых испытаний пилотной системы, прежде чем приступить к поставке реального продукта. [Теперь, с появлением бета-версий, это стало обычной практикой. Бета-версия — не то же самое, что прототип с ограниченной функциональностью, альфа-версия, в защиту которого я бы также выступил.]

11.3. В большинстве проектов первая построенная система практически не используется: она — слишком медленная, слишком большая, слишком сложная в использовании или всё вместе взятое.

11.4. Отбраковка и редизайн могут быть сделаны одним махом или же по частям. Неважно как, но *это все равно будет сделано*.

11.5. Релиз первой системы, одноразовой, поможет выиграть время, но только за счет агонии пользователя. Побочными эффектами

можно назвать и отвлечение внимания строителей, поддерживающих систему в период ее редизайна, и закрепление плохой репутации для продукта, которую будет трудно пережить.

11.6. Поэтому *планируйте выбросить первую версию: вы все равно это сделаете.*

11.7. «Программист поставляет удовлетворение потребности пользователя, а не какой-то осязаемый продукт» (Косгроув).

11.8. Как фактическая потребность, так и восприятие этой потребности пользователем будут *меняться* по мере разработки, тестирования и использования программ.

11.9. Гибкость и невидимость программного продукта подвергают его разработчиков (исключительно) воздействию постоянных изменений в требованиях.

11.10. Некоторые обоснованные изменения в целевых критериях (и в стратегиях развития) неизбежны, и лучше быть к ним готовым, чем исходить из того, что они не наступят.

11.11. Методика планирования программного продукта с учетом будущего изменения, особенно структурное программирование с тщательной документацией интерфейсов модулей, хорошо известна, но не всегда практикуется. Она также помогает использовать табличные методы везде, где это возможно. [Современные затраты на память и размеры постоянно улучшают такую методику.]

11.12. Следует использовать язык высокого уровня, операции времени компиляции, встраивание объявлений по ссылке и самодокументированную методику с целью уменьшения ошибок, индуцируемых изменением.

11.13. Следует квантовать изменения в четко определенных пронумерованных версиях. [Теперь стандартная практика.]

ПЛАНИРОВАНИЕ ОРГАНИЗАЦИИ С УЧЕТОМ БУДУЩИХ ИЗМЕНЕНИЙ

11.14. Нежелание программиста документировать проекты исходит не столько от лени, сколько от нерешительности в принятии на себя защиты решений, которые, как известно дизайнеру, носят предварительный характер (Косгроув).

11.15. Структурировать организацию с учетом будущего изменения гораздо сложнее, чем спланировать систему с учетом такого изменения.

11.16. Начальник проекта должен работать над тем, чтобы менеджеры и технические работники были взаимозаменяемы настолько, насколько позволяют их таланты; в частности, идеально иметь возможность легко перемещать людей между техническими и управленческими ролями.

11.17. Барьеры на пути эффективной организации с двойной служебной лестницей носят социологический характер, и с ними необходимо бороться с постоянной бдительностью и энергией.

11.18. Для соответствующих ступеней двойной служебной лестницы легко установить соответствующие шкалы окладов, но это требует решительных инициативных мер для придания им соответствующего престижа: равных должностей, равных вспомогательных услуг, сверхкомпенсирующих управленческих действий.

11.19. Организация по принципу хирургической бригады — это радикальная атака на все аспекты данной проблемы. Она действительно дает долгосрочный ответ на задачу по созданию гибкой организации.

ДВА ШАГА ВПЕРЕД И ОДИН НАЗАД — СОПРОВОЖДЕНИЕ ПРОГРАММЫ

11.20. Сопровождение программного обеспечения принципиально отличается от сопровождения аппаратного обеспечения. Оно состоит главным образом из изменений, которые устраняют ошибки конструкции, добавляют инкрементную функциональность или адаптируют к изменениям в среде использования или конфигурации.

11.21. Суммарная стоимость сопровождения в течение жизни широко используемой программы, как правило, составляет 40 % или более от стоимости ее разработки.

11.22. Стоимость сопровождения строго зависит от числа пользователей. Чем больше пользователей, тем больше ошибок они находят.

11.23. Кэмпбелл отмечает интересную кривую взлета и падений обнаруживаемых ошибок в течение жизни продукта.

11.24. Исправление ошибки имеет существенный (от 20 до 50 %) шанс внесения еще одной.

11.25. После каждого исправления необходимо провести полное регрессионное тестирование системы и убедиться, что она не была повреждена непонятным образом.

11.26. Методы дизайна программ, когда устраняются или, по крайней мере, освещаются побочные эффекты, могут иметь огромную отдачу в стоимости сопровождения.

11.27. То же самое можно сказать и о методах имплементации дизайнов меньшим числом людей, меньшим числом интерфейсов и с меньшим числом ошибок.

Один шаг вперед и один шаг назад — энтропия системы возрастает в течение жизни.

11.28. Леман и Беладди обнаружили, что общее число модулей линейно увеличивается с числом релизов крупной операционной системы (OS/360), но при этом число затронутых модулей увеличивается экспоненциально с числом релизов.

11.29. Все изменения имеют тенденцию разрушать структуру, увеличивать энтропию и беспорядок системы. Даже самое умелое сопровождение программы лишь задерживает ее погружение в неразрешимый хаос, из которого должен быть сделан основательный редизайн. [Иногда реальная необходимость обновления программы, например, с целью повышения производительности, вызывает необходимость изменения внутренних границ структур. Часто первоначальные границы становились причиной недостатков, которые позже всплывали на поверхность.]

ГЛАВА 12. ЗАТОЧЕННЫЕ ИНСТРУМЕНТЫ

12.1. Менеджеру проекта необходимо выработать философию и выделить ресурсы для строительства общих инструментов, а также признать необходимость использования персонализированных инструментов.

12.2. Командам, строящим операционные системы, нужна собственная целевая машина для отладки; ей требуется максимальная память, а не максимальная скорость и системный программист, чтобы поддерживать стандартное программное обеспечение в рабочем состоянии.

12.3. Отладочная машина или ее программное обеспечение также должны быть инструментированы для того, чтобы можно было

вести подсчет и измерение всех видов параметров программы в автоматическом режиме.

12.4. Требование к использованию целевой машины имеет своеобразную кривую роста: низкая активность, за которой следует взрывной рост, и затем выравнивание.

12.5. Отладка системы, как и исследования в астрономии, всегда проводится в основном ночью.

12.6. Выделение значительных блоков времени целевой машины на одну подкоманду за один раз оказалось лучшим способом планирования временного графика, гораздо лучшим, чем ее использование чередующимися подкомандами, несмотря на теорию.

12.7. Этот предпочтительный метод поблочного планирования графика использования времени дефицитных компьютеров пережил 20 лет [в 1975 году] технологических изменений, потому что является наиболее продуктивным. [В 1995 году это по-прежнему так.]

12.8. Если целевой компьютер является новым, то для него требуется логический эмулятор. Его получают как можно *раньше*, и он обеспечивает *надежный* отладочный носитель даже после появления нестоящей машины.

12.9. Библиотека мастер-программ должна быть разделена на: (1) набор отдельных песочниц; (2) подбиблиотеку системной интеграции, в текущее время находящуюся в состоянии системного тестирования; и (3) выпущенную версию. Формальное разделение и прогрессия обеспечивают контроль.

12.10. Инструментом, который экономит больше всего трудозатрат в проекте программирования, вероятно, является текстовый редактор.

12.11. Объемность в системной документации действительно вводит новый вид непостижимости [см. Unix, например], но это

гораздо предпочтительнее, чем недостаточное документирование, которое так распространено.

12.12. Создайте эмулятор производительности снаружи внутрь, сверху вниз. Начните работу с ним как можно раньше. Прислушайтесь к тому, что он вам скажет.

ЯЗЫК ВЫСОКОГО УРОВНЯ

12.13. Только лень и инертность препятствуют всеобщему внедрению языка высокого уровня и интерактивного программирования. [И сегодня они были приняты повсеместно.]

12.14. Язык высокого уровня повышает не только продуктивность, но и культуру разработки: программа имеет меньше ошибок, и их проще находить.

12.15. Классические возражения относительно функциональности, пространства объектного кода и скорости объектного кода стали неактуальными в связи с развитием языка и компиляторной технологии.

12.16. Единственным разумным кандидатом на системное программирование сегодня является PL/1. [Больше не соответствует действительности.]

ИНТЕРАКТИВНОЕ ПРОГРАММИРОВАНИЕ

12.17. Интерактивные системы никогда не заменят пакетные системы для некоторых приложений. [По-прежнему верно.]

12.18. Отладка является трудоемкой и времязатратной частью системного программирования, а медленный оборот — проклятием отладки.

12.19. Ограниченные свидетельства показывают, что интерактивное программирование как минимум удваивает продуктивность системного программирования.

ГЛАВА 13. ЦЕЛОЕ И ЧАСТИ

13.1. Подробные, кропотливые архитектурные усилия, описанные в главах 4, 5 и 6, упрощают не только использование продукта, но и его разработку, а также сокращают число системных ошибок, которые необходимо отыскивать.

13.2. Высоцкий говорит: «Многие, очень многие неудачи касаются именно тех аспектов, которые никогда не были точно определены».

13.3. Задолго до появления самого кода спецификация должна быть передана внешней группе тестирования для тщательного изучения на предмет полноты и ясности. Сами разработчики не могут этого сделать. (Высоцкий)

13.4. «Нисходящий дизайн Вирта [путем пошагового уточнения] является наиболее важной новой формализацией программирования всего десятилетия [1965– 1975]».

13.5. Вирт выступает за использование как можно более высокого уровня нотации на каждом шаге.

13.6. У хорошего нисходящего дизайна есть четыре пути, чтобы избежать ошибок.

13.7. Иногда приходится отступить назад, отказываться от высокой планки и начинать все сначала.

13.8. Структурное программирование, то есть разработка программ, управляющие структуры которых состоят только из заданного набора операторов, воздействующих на блоки кода (в отличие от

разнообразного ветвления), является здоровым способом избежать ошибок и демонстрирует правильный образ мыслей.

13.9. Экспериментальные результаты Голда показывают, что в интерактивной отладке достигается в три раза больше прогресса при первом взаимодействии каждого сеанса, чем при последующих взаимодействиях. По-прежнему стоит тщательно планировать отладку перед входом в систему. [Я думаю, что в 1995 году это *все еще так*.]

13.10. Надлежащее использование хорошей системы [интерактивная отладка с быстрым откликом] требует двух часов за столом для каждого двухчасового сеанса на машине: один час на уборку и документирование после сеанса и один час на планирование изменений и тестов для следующего раза.

13.11. Отладка системы (в отличие от отладки компонентов) займет больше времени, чем ожидается.

13.12. Усилия, затраченные на отладку системы, оправдывают тщательно систематизированный и плановый подход.

13.13. Следует начинать отладку системы только после того, как ее части, по-видимому, работают (в отличие от подхода «прикрутить все вместе и попробовать» с целью выкурить интерфейсные ошибки и в отличие от запуска отладки системы, когда компонентные ошибки полностью известны, но не исправлены.) [Это особенно верно для команд.]

13.14. Следует возводить как можно больше отладочных строительных лесов и тестового кода, возможно, в размере вплоть до 50 % отлаживаемого продукта.

13.15. Необходимо контролировать и документировать изменения и версии, при этом члены команды работают с копиями в песочницах.

13.16. Во время отладки системы следует добавлять по одному компоненту за раз.

13.17. Леман и Белادي предлагают доказательства того, что кванты изменения должны быть большими и нечастыми или же очень малыми и частыми. Последнее в большей степени подвержено нестабильности. [Команда Microsoft делает небольшие частые кванты. Растущая система перестраивается каждую ночь.]

ГЛАВА 14. И ГРЯНУЛ ГРОМ

14.1. Как оказывается, что проект запаздывает на один год? Сначала он запаздывает на один день.

14.2. Ежедневное отставание от графика трудно распознать, еще труднее предотвратить и труднее компенсировать.

14.3. Первым шагом в управлении крупным проектом по жесткому графику является *планирование* графика, состоящего из дат контрольных точек.

14.4. Контрольные точки должны быть конкретными, определяемыми, измеримыми событиями, словно зарубки, сделанные острым ножом.

14.5. Программист редко будет лгать о продвижении контрольных точек, если контрольная точка является настолько четкой, что обмануться невозможно.

14.6. Исследования с оцениванием поведения правительственных подрядчиков на крупномасштабных строительных проектах показывают, что оценки времени деятельности, тщательно пересматриваемые каждые две недели, существенно не изменяются по мере приближения начала работ; что в ходе деятельности завышенные

оценки неуклонно снижаются, а *заниженные* оценки не изменяются примерно за 3 недели до запланированного завершения.

14.7. Хроническое отставание графика является убийцей морали. [Джим Маккарти (Jim McCarthy) из Microsoft говорит: «Если вы пропустите один дедлайн, то непременно пропустите и следующий».]

14.8. Для великих команд по программированию *задиристость* необходима так же, как и для великих бейсбольных команд.

14.9. Ничто не может заменить сетевой график критического пути, который позволяет отличать, какие смещения и насколько имеют значение.

14.10. Подготовка диаграммы критического пути является наиболее ценной частью ее использования, поскольку строительство сети, выявление зависимостей и оценка сегментов требуют большого количества очень специфичного планирования на самом раннем этапе проекта.

14.11. Первая диаграмма всегда является ужасной, и требуются некоторые творческие усилия, чтобы создать следующую.

14.12. Диаграмма критического пути отвечает на деморализующее оправдание: «другая часть работы все равно запаздывает».

14.13. Каждому начальнику нужны два вида данных: информация о срывах сроков, которая требует вмешательства, и картина состояния дел, чтобы быть осведомленным и иметь раннее предупреждение.

14.14. Получить информацию о состоянии дел сложно, так как менеджеры имеют все основания не делиться подобной информацией.

14.15. Неверным действием босс может гарантировать полное раскрытие информации о состоянии дел; и наоборот, тщательное

разделение отчетов о состоянии дел и принятие их без паники или предвосхищения будет способствовать честной отчетности.

14.16. Необходимо иметь обзорную методику, с помощью которой истинное состояние дел становится известным всем игрокам. Ключевыми для этой цели являются деление графика на контрольные точки и документ о завершении работ.

14.17. Высоцкий: «Я нашел удобным переносить “запланированные” в графике даты (даты начальника) и “оценочные” даты (даты менеджера низшего звена) в отчет о контрольной точке. Менеджер проекта должен держаться подальше от оценочных дат».

14.18. Небольшая группа «планирования и осуществления контроля», которая ведет отчет по контрольным точкам, имеет неоценимое значение для крупного проекта.

ГЛАВА 15. ДРУГАЯ СТОРОНА

15.1. Другая сторона программного продукта, обращенная к пользователю, документация, в полной мере так же важна, как и сторона, обращенная к машине.

15.2. Даже для самых частных программ необходима текстовая документация, ибо память непременно подведет пользователя-автора.

15.3. Преподавателям и менеджерам в целом не удалось привить программистам такое отношение к документации, которое будет вдохновлять их на всю жизнь, преодолевая лень и давление со стороны графика.

15.4. Эта неудача объясняется не столько отсутствием рвения или красноречия, сколько неспособностью показать, *как* эффективно и экономно нужно документировать.

15.5. Документация часто страдает отсутствием общего обзора. Посмотрите сначала издалека, а потом медленно приближайтесь.

15.6. Критически важная документация пользователя должна быть подготовлена до того, как программа будет построена, поскольку она воплощает основные планировочные решения. Она должна описывать девять вещей (см. главу).

15.7. Программа должна поставляться с несколькими тестовыми случаями, часть из них предназначены для допустимых входных данных, часть для граничных входных данных, и часть для явно недопустимых входных данных.

15.8. Документация внутренних частей программы для ее модификатора также требует текстового обзора, который должен содержать пять видов вещей (см. главу).

15.9. Блок-схема является чересчур переоцененным фрагментом документации по программированию; подробная блок-схема является неприятностью, не актуальной в связи с развитием языков высокого уровня. (Блок-схема — это *схематичный* язык высокого уровня.)

15.10. Не многие программы нуждаются в более чем одностраничной блок-схеме. [Требования к документации MILSPEC действительно неверны в этом вопросе.]

15.11. В действительности требуется структурный граф программы, который не нуждается в стандартах ANSI строительства блок-схем.

15.12. Для поддержания документации в актуальном состоянии крайне важно, чтобы она встраивалась в исходную программу, а не хранилась в виде отдельного документа.

15.13. Ключевыми для минимизации бремени документирования являются следующие три понятия:

- Как можно больше используйте для документирования обязательные части программы, такие как имена и объявления.
- Используйте пространство и формат для показа подчиненности и вложенности, а также улучшения читаемости.
- Вставляйте необходимую текстовую документацию в программу в виде абзацев комментариев, а еще лучше — в виде модульных заголовков.

15.14. В документации, которой будут пользоваться при модификации программы, объясняйте не только «как», но и «почему». Назначение является решающим для понимания. Даже языки высокого уровня совсем не передают значения.

15.15. Методика самодокументированного программирования находит свое наибольшее применение и силу в языках высокого уровня, используемых с онлайн-системами, являющимися инструментами, которые обязательно следует использовать.

ЭПИЛОГ ПЕРВОГО ИЗДАНИЯ

Э.1. Программные системы, возможно, являются самыми замысловатыми и многосложными (с точки зрения числа различных видов частей) из всего, что делает человечество.

Э.2. «Смоляная яма» инженерии программного обеспечения будет оставаться вязкой еще долгое время.



19

МИФИЧЕСКИЙ ЧЕЛОВЕКО- МЕСЯЦ ДВАДЦАТЬ ЛЕТ СПУСТЯ

Я не знаю другого способа судить
о будущем, как с помощью прошлого.

Патрик Генри

Опираясь на прошлое, невозможно
планировать будущее.

Эдмунд Берк

ЗАЧЕМ В 1995 ГОДУ ПОНАДОБИЛОСЬ ПЕРЕИЗДАВАТЬ КНИГУ?

Самолет гудел в ночи, направляясь к аэропорту «Ла-Гуардия». Облака и темнота скрывали все интересные достопримечательности. Документ, который я изучал, был неинтересным. Однако мне не было скучно. Незнакомец, сидевший рядом со мной, читал «Мифический человеко-месяц», и я ждал, как он отреагирует — словом или знаком. Наконец, когда мы вырулили к выходу, я не выдержал:

— Как вам эта книга? Рекомендуете?

— Гм! Ничего такого, чего бы я уже не знал.

Я предпочел не представляться.

Почему «Мифический человеко-месяц» продолжает жить? Почему эта книга по-прежнему считается актуальной для практики программного обеспечения сегодня? Почему она имеет читательскую аудиторию за пределами сообщества инженерии программного обеспечения, генерируя обзоры, цитаты и переписку от юристов, врачей, психологов, социологов, а также от людей, занимающихся программным обеспечением? Как может книга, написанная 20 лет назад об опыте сборки программного обеспечения 30 лет назад, все еще быть актуальной, а тем более полезной?

Одно из объяснений, которое иногда можно услышать, состоит в том, что дисциплина разработки программного обеспечения не продвинулась должным образом. Эта точка зрения часто подкрепляется противопоставлением продуктивности разработки компьютерных программ продуктивности производства компьютерного оборудования, которая за два десятилетия увеличилась по меньшей мере в тысячу раз. Как поясняется в главе 16, аномалия

заключается не в том, что программное обеспечение развивалось так медленно, а в том, что компьютерная технология претерпела взрывной рост способом, не имеющим аналогов в истории человечества. По большому счету, он проистекает от постепенного перехода компьютерного производства из сборочной индустрии в обрабатывающую, из трудоемкого к капиталоемкому производству. Разработка аппаратного и программного обеспечения, в отличие от производства, остается, в сущности, трудоемкой.

Второе объяснение, которое часто выдвигается, заключается в том, что «Мифический человеко-месяц» только эпизодически связан с программным обеспечением, но в первую очередь с тем, как люди в командах делают продукт. Несомненно, в этом есть доля правды; в предисловии к изданию 1975 года говорится, что менеджмент проекта по разработке программного обеспечения больше похож на любой другой менеджмент, чем большинство программистов изначально полагают. Я все еще верю, что это правда. Человеческая история — это драма, в которой сюжеты остаются неизменными, сценарии этих сюжетов меняются медленно вместе с развитием культур, а сценические декорации меняются все время. Таким образом, мы видим себя в двадцатом веке отраженными в Шекспире, Гомере и Библии. Поэтому в той мере, в какой «Мифический человеко-месяц» написан о людях и командах, он устаревает медленно.

Какова бы ни была причина, читатели продолжают покупать эту книгу, и они продолжают присылать мне комментарии, которые я очень ценю. В наши дни меня часто спрашивают: «Как вы считаете, в чем вы тогда ошиблись? Что сейчас устарело? Что действительно нового в мире инженерии программного обеспечения?» Все эти совершенно разные вопросы справедливы, и я постараюсь ответить на них как можно полнее. Однако не в таком порядке, а в тематических кластерах. Во-первых, давайте рассмотрим то, что было правильным во времена создания книги и по-прежнему остается таковым.

ЦЕНТРАЛЬНЫЙ АРГУМЕНТ: КОНЦЕПТУАЛЬНАЯ ЦЕЛОСТНОСТЬ И АРХИТЕКТОР

КОНЦЕПТУАЛЬНАЯ ЦЕЛОСТНОСТЬ. Чистый, элегантный программный продукт должен представлять каждому из своих пользователей когерентную ментальную модель приложения, стратегии выполнения приложения и тактику пользовательского интерфейса, которая будет использоваться при детализировании действий и параметров. Концептуальная целостность продукта, как он воспринимается пользователем, является наиболее важным фактором в простоте использования. (Конечно, есть и другие факторы. Единообразие пользовательского интерфейса во всех приложениях Macintosh является важным примером. К тому же можно создавать когерентные интерфейсы, которые, тем не менее, достаточно угловатые. Возьмите MS-DOS.)

Есть много примеров элегантных программных продуктов, разработанных одним человеком или двумя. Так создаются большинство чисто интеллектуальных произведений, таких как книги или музыкальные композиции. Однако процессы разработки продуктов во многих индустриях не могут позволить себе такой простой подход к концептуальной целостности. Конкурентное давление вынуждает к срочности; во многих современных технологиях конечный продукт является довольно многосложным, и дизайн по своей сути требует многих человеко-месяцев усилий. Программные продукты одновременно являются сложными и жестко конкурентными по графику выполнения работ.

Любой продукт, который является настолько большим или срочным, что требует усилий многих умов, таким образом, сталкивается с особой трудностью: результат должен концептуально согласовываться с разумом одиночного пользователя и в то же время проектироваться усилиями нескольких разумов. Как организовать проектные усилия для достижения такой концептуальной целостности?

Один из ее тезисов заключается в том, что управление большими проектами качественно отличается от управления малыми просто из-за числа вовлеченных умов. Для достижения когерентности необходимы целенаправленные и даже героические действия со стороны менеджмента.

АРХИТЕКТОР. В главах с четвертой по шестую я доказываю, что самое важное — назначить одного человека *архитектором продукта*, ответственным за все его стороны, воспринимаемые пользователем. Пусть этот человек отвечает за концептуальную целостность всех аспектов продукта, воспринимаемых пользователем. Архитектор формирует и имеет в своем владении общедоступную идеальную модель продукта, с помощью которой пользователю будет объяснено его применение. Сюда входит подробная спецификация всей его функциональности и средств для ее вызова и управления. Архитектор также является агентом пользователя, со знанием дела представляя его интерес в неизбежных компромиссах между функциональностью, производительностью, размером, стоимостью и графиком. Эта роль является работой на полный день, и только в самых малых командах она может быть совмещена с работой менеджера команды. Архитектор подобен режиссеру, а менеджер — продюсеру кинофильма.

ОТДЕЛЕНИЕ АРХИТЕКТУРЫ ОТ ИМПЛЕМЕНТАЦИИ И РЕАЛИЗАЦИИ. Для того чтобы важнейшая работа архитектора началась хотя бы в мыслях, необходимо отделить архитектуру, определение продукта как он воспринимается пользователем, от ее имплементации. Архитектура и разработка определяют четкую грань между разными частями задачи проектирования, и по каждую сторону этой грани лежит большая работа.

РЕКУРСИЯ АРХИТЕКТОРОВ. В очень больших проектах одному человеку не справиться со всей архитектурой, даже если он избавлен от всех забот, связанных с реализацией. Поэтому необходимо, чтобы главный архитектор системы разделил систему на подсистемы.

Границы подсистем легче всего строго определить в тех местах, где интерфейсы между подсистемами минимальны. Тогда каждая часть будет иметь своего собственного архитектора, который должен отчитываться перед главным архитектором системы в отношении архитектуры. Очевидно, что этот процесс может продолжаться рекурсивно по мере необходимости.

СЕГОДНЯ Я УБЕЖДЕН БОЛЬШЕ, ЧЕМ КОГДА-ЛИБО. Концептуальная целостность является центральной для качества продукта. Наличие системного архитектора — самый важный шаг на пути к концептуальной целостности. Эти принципы ни в коем случае не ограничиваются программными системами, а относятся к планированию любого сложного конструкта, будь то компьютер, самолет, стратегическая оборонная инициатива или глобальная система позиционирования. После 20 лет преподавания в лаборатории инженерии программного обеспечения я стал настаивать на том, чтобы студенческие команды из четырех человек выбирали менеджера и отдельного архитектора. Определение различных ролей в таких командах, возможно, выглядит немного экстремальным, но я заметил, что это хорошо работает и способствует успеху дизайна даже для малых команд.

ЭФФЕКТ ВТОРОЙ СИСТЕМЫ: ПОЛЗУЧИЙ «УЛУЧШИЗМ» И УГАДЫВАНИЕ ЧАСТОТ

ДИЗАЙН ДЛЯ ОБШИРНОЙ АУДИТОРИИ ПОЛЬЗОВАТЕЛЕЙ. Одним из последствий революции персональных компьютеров является то, что все чаще, по крайней мере в сообществе обработки бизнес-данных, коммерческие программные продукты заменяют заказные приложения. Более того, стандартные программные пакеты продаются сотнями тысяч копий, а то и миллионами. Системные архитекторы программ, поставляемых вместе с машиной, всегда должны были планировать проект для большого, аморфного множества поль-

зователей, а не для одного определяемого приложения в одной компании. Многие, очень многие системные архитекторы сталкиваются с этой работой.

Как это ни парадоксально, но спроектировать инструмент общего назначения гораздо труднее, чем инструмент специального назначения, именно потому, что приходится назначать веса разным потребностям различных пользователей.

ПОЛЗУЧИЙ «УЛУЧШИЗМ». Непреодолимое искушение для архитектора инструмента общего назначения, такого как электронная таблица или текстовый процессор, состоит в перегрузке продукта функциями незначительной полезности за счет производительности и даже простоты использования. Привлекательность предлагаемых функций очевидна с самого начала; снижение в производительности становится очевидным только по мере продолжения тестирования системы. Потеря простоты использования подкрадывается коварно, по мере того как фичи добавляются небольшими инкрементами и документация становится все толще и толще.¹

Для продуктов массового рынка, которые выживают и развиваются на протяжении многих поколений, этот соблазн особенно силен. Миллионы клиентов запрашивают сотни функциональных свойств; любой запрос сам по себе является доказательством того, что «рынок этого требует». Часто архитектор первоначальной системы уже ушел в поход за новой славой, и архитектура оказалась в руках людей с меньшим опытом взвешенного представления общих интересов пользователей. В недавнем обзоре Microsoft Word 6.0 говорится: «Word 6.0 упакован функционалом под завязку; обновление замедлено трудностями обеспечения совместимости с предыдущими версиями... Word 6.0, к тому же, большой и медленный». В нем с тревогой отмечается, что Word 6.0 требует 4 Мб оперативной памяти, и при этом нас убеждают, что насыщенная добавленная функциональность означает: «даже Macintosh IIx едва дотягивает до уровня работы Word 6».²

ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ПОЛЬЗОВАТЕЛЕЙ. Чем больше и аморфнее множество пользователей, тем более необходимо явное определение общего портрета пользователя для достижения концептуальной целостности. Каждый член команды разработчиков, несомненно, будет иметь неявный образ пользователей, и образ каждого разработчика будет отличаться. Поскольку представление архитектора о пользователе сознательно или подсознательно влияет на каждое архитектурное решение, крайне важно, чтобы команда разработчиков пришла к общему образу. И это требует записи атрибутов ожидаемого множества пользователей, в том числе:

- Кто они такие.
- Что им нужно.
- Что, по их мнению, им нужно.
- Чего они хотят.

ЧАСТОТЫ. Для любого программного продукта любой из атрибутов множества пользователей фактически является распределением со многими возможными значениями, каждое из которых имеет свою частоту. Как архитектор должен прийти к этим частотам? Обследование этой слабо определенной популяции является сомнительным и дорогостоящим делом.³ С годами я пришел к убеждению, что архитектор должен *угадывать* или, если угодно, *предполагать* полное множество атрибутов и значений с их частотами, для того чтобы развить полное, явное и общее для всех описание группы пользователей.

Из этой редко происходящей процедуры вытекают многие выгоды. Во-первых, процесс угадывания частот заставит архитектора очень тщательно подумать об ожидаемом множестве пользователей. Во-вторых, при фиксации частот возникает обсуждение, полезное для всех участников и выявляющее различия в образах пользователя, имеющих у разных разработчиков. В-третьих, перечисление частот явно помогает каждому понять, какие решения от каких множеств пользователей зависят. Ценен даже такой неформаль-

ный анализ чувствительности. Когда выясняется, что очень важные решения зависят от какой-то конкретной догадки, то затраты стоят того, чтобы установить для этой стоимости более точные оценки. (Система gIBIS, разработанная Джеффом Конклином (Jeff Conklin), предоставляет инструмент для формального и точного отслеживания дизайнерских решений и документирования причин каждого из них.⁴ У меня не было возможности его использовать, но думаю, что это было бы очень полезно.)

Подведем итог: запишите явные догадки в отношении атрибутов множества пользователей. *Гораздо лучше ошибаться, но говорить прямо, чем выражаться расплывчато.*

ЧТО НАСЧЕТ «ЭФФЕКТА ВТОРОЙ СИСТЕМЫ»? Один проницательный студент заметил, что «Мифический человеко-месяц» предлагает рецепт катастрофы: он состоит в планировании доставки второй версии любой новой системы (глава 11), которую глава 5 характеризует как наиболее опасную запланированную систему. Пришлось признать, что он подловил меня.

Указанное противоречие является скорее лингвистическим, чем реальным. «Вторая» система, описанная в главе 5, — это вторая выпущенная система, последующая система, которая привлекает добавленную функциональность и излишества. «Вторая» система в главе 11 — это вторая попытка построить то, что должно быть первой выпущенной системой. Она построена в соответствии со всеми ограничениями со стороны графика, таланта и невежества, которые характеризуют новые проекты, — ограничения, навязывающие дисциплину умеренности.

ТРИУМФ ИНТЕРФЕЙСА WIMP

Одним из самых впечатляющих достижений в инженерии ПО за последние два десятилетия стал триумф интерфейса окон, иконок,

меню, курсора, или сокращенно WIMP (от англ. Windows, Icons, Menus, Pointing). Сегодня он стал настолько привычным, что не нуждается в описании. Эта концепция была впервые публично продемонстрирована Дугом Энглбартом (Doug Englebart) и его командой из Стэндфордского исследовательского института на Объединенной компьютерной конференции Запада (Western Joint Computer Conference) 1968 года.⁵ Оттуда идеи пошли в исследовательский центр Xerox Palo Alto, где они появились в персональной рабочей станции Alto, разработанной Бобом Тейлором (Bob Taylor) и командой. Они были подобраны Стивом Джобсом для Apple Lisa, компьютера слишком медленного, чтобы вынести его захватывающие концепции простоты использования. Эти концепции Джобс затем воплотил в коммерчески успешном Apple Macintosh в 1985 году. Позже они были приняты в Microsoft Windows для IBM PC и совместимых компьютеров. Версия Mac будет моим примером.⁶

КОНЦЕПТУАЛЬНАЯ ЦЕЛОСТНОСТЬ ЧЕРЕЗ МЕТАФОРУ. WIMP — это превосходный пример пользовательского интерфейса, который обладает концептуальной целостностью, достигнутой путем принятия хорошо знакомой ментальной модели, метафоры рабочего стола, и ее тщательного последовательного развития для использования воплощения в компьютерной графике. Например, из принятой метафоры непосредственно следует сложно осуществимое, но правильное решение о перекрытии окон вместо расположения их одно рядом с другим. Возможность изменять размер и форму окон является последовательным расширением, которое дает пользователю новые возможности, ставшие доступными благодаря среде компьютерной графики. Подобное не так просто совершить с настоящими бумагами на настоящем рабочем столе. Перетаскивание следует непосредственно из метафоры; выбор иконок, посредством указывания на них курсором, является прямым аналогом выбора вещей рукой. Иконки и вложенные папки являются верными аналогами настольных документов, так же как и корзина. Концепции

вырезания, копирования и вставки точно отражают то, что мы привыкли делать с документами за рабочим столом. Метафора настолько точно повторяется и настолько последовательно ее расширение, что новые пользователи несомненно испытывают шок от идеи перетаскивания значка дискеты в корзину, для того чтобы извлечь диск. Если бы интерфейс не был почти равномерно последовательным, то эта (довольно плохая) несогласованность раздражала бы не так сильно.

В каких местах интерфейс WIMP вынужден далеко отойти от метафоры рабочего стола? Наиболее заметны два отличия: меню и работа одной рукой. При работе с реальным рабочим столом человек *выполняет* действия с документами, а не говорит кому-то или чему-то делать это. И когда все-таки кто-то говорит кому-то выполнить действие, он обычно генерирует, а не выбирает устные или письменные команды с глаголом в повелительном наклонении: «Пожалуйста, подайте это», «Пожалуйста, найдите предыдущую корреспонденцию», «Пожалуйста, отправьте это Мэри, чтобы она этим занялась».

Увы, достоверная интерпретация произвольных английских команд выходит за рамки современного уровня техники, будь то письменные или устные команды. И поэтому разработчики интерфейса на два шага удалились от прямого действия пользователя с документами. Они благоразумно переняли метафору выбора команды с обычного рабочего стола — печатный бланк, где из ограниченного меню пользователь делает выбор команд, семантика которых стандартизирована. Эту идею они распространили на горизонтальное меню вертикальных выпадающих подменю.

КОМАНДНЫЕ ИМПЕРАТИВЫ И ПРОБЛЕМА С ДВУМЯ КУРСОРАМИ. Команды — это императивные (повелительные) предложения; они всегда имеют глагол и обычно имеют прямой объект (косвенное дополнение). Для любого действия нужно указать глагол и существительное. Указательная метафора говорит, что для того чтобы указывать






две вещи одновременно, следует иметь два отдельных курсора на экране, каждый из которых управляется отдельной мышью — одной в правой руке и одной в левой. Ведь на физическом рабочем столе мы обычно работаем обеими руками. (Но одна рука часто держит вещи в фиксированном состоянии, что происходит по умолчанию на рабочем столе компьютера.) Ум, безусловно, способен работать двумя руками; мы регулярно используем две руки при печатании, вождении, приготовлении пищи. Увы, предоставление одной мыши было уже большим шагом вперед для производителей персональных компьютеров; ни одна коммерческая система не приспособлена для совершения одновременных действий двумя курсорами мыши — по одному в каждой руке.

Разработчики смирились с реальностью и спланировали поведение для одной мыши, приняв синтаксическое соглашение, что сначала нужно указать (*выбрать*) существительное. И нужно указать глагол, пункт меню. При этом в значительной мере утрачивается простота использования. Когда я смотрю на пользователей, или видеозаписи пользователей, или компьютерные трассировки движений курсора, меня поражает, что один курсор должен выполнять работу за двоих: выбрать объект в рабочей части окна; выбрать глагол в меню; найти или повторно найти объект на рабочем столе; снова вытащить меню (часто одно и то же) и выбрать глагол. Курсор перемещается туда-сюда, туда-сюда, из пространства данных в пространство меню, каждый раз выбрасывая полезную информацию о том, где он был в последний раз в этом пространстве, — в целом неэффективный процесс.


БЛЕСТЯЩЕЕ РЕШЕНИЕ. Даже если электроника и программное обеспечение могут легко обрабатывать два одновременно активных курсора, существуют трудности с пространственным расположением. Рабочий стол в метафоре WIMP реально включает пишущую машинку, и нужно разместить ее реальную клавиатуру в физическом пространстве на реальном рабочем столе. Клавиатура плюс два коврика для мыши используют большую часть пространства

в пределах досягаемости руки. Так вот, проблема клавиатуры может быть превращена в потенциал — почему бы не задействовать эффективную работу двумя руками, одной рукой задавая на клавиатуре глаголы, а другой рукой выбирая существительные с помощью мыши. Теперь курсор остается в пространстве данных, эксплуатируя высокую локальность поочередного выбора существительных. Реальная эффективность, реальная сила пользователя.

СИЛА ПОЛЬЗОВАТЕЛЯ ПРОТИВ ПРОСТОТЫ В ИСПОЛЬЗОВАНИИ. Однако при таком решении теряется то, что делает использование меню таким простым для новичков, — меню представляют альтернативные глаголы, допустимые в любом конкретном состоянии. Мы можем купить программный пакет, принести его домой и начать использовать его, не обращаясь к справочному руководству, просто зная, для чего мы его купили, и экспериментируя с разными глаголами меню.

Одна из самых сложных проблем, стоящих перед архитекторами ПО, заключается именно в том, как сбалансировать силу пользователя и простоту использования. Нужно ли проектировать программу в расчете на новичка и случайного пользователя или строить ее с мощными функциями для профессионала? Идеальный ответ в том, чтобы обеспечить оба подхода концептуально согласованным образом, — это достигается в интерфейсе WIMP. Высокочастотные глаголы меню имеют эквиваленты, состоящие из пар «отдельная клавиша + командная клавиша», в основном выбранные так, чтобы их можно было легко нажимать, как один аккорд, левой рукой. Например, на компьютере Mac командная клавиша (||) находится чуть ниже клавиш Z и X, поэтому самые высокочастотные операции кодируются как  z,  x,  c,  v,  s.

ИНКРЕМЕНТНЫЙ ПЕРЕХОД ОТ НОВИЧКА К ОПЫТНОМУ ПОЛЬЗОВАТЕЛЮ. Эта двойная система указания командных глаголов не только удовлетворяет потребности новичка в малом усвоении информации и потребности опытного пользователя в эффективности, но и обеспе-

чивает каждому пользователю плавный переход между режимами. Кодировки букв, именуемые *горячими клавишами* (шорткатами), отображаются в меню рядом с глаголами, так что сомневающийся пользователь может вытянуть меню вниз и проверить буквенный эквивалент, а не просто выбрать пункт меню. Каждый новичок сначала изучает горячие клавиши для своих собственных частых операций. Любую горячую клавишу, в которой он сомневается, он может попробовать, так как  z отменит любую оплошность. Кроме того, он может проверить меню и увидеть, какие команды являются допустимыми. Новички будут часто обращаться к меню, опытные пользователи — очень мало; и промежуточным пользователям только иногда нужно будет выбрать из меню, так как каждый будет знать несколько горячих клавиш, которые составляют большинство его собственных операций. Большинство из нас, дизайнеров ПО, слишком хорошо знакомы с этим интерфейсом, чтобы в полной мере оценить его элегантность и мощь.

УСПЕХ ПРЯМОГО ВСТРАИВАНИЯ КАК ПРИЕМА ДЛЯ СОБЛЮДЕНИЯ АРХИТЕКТУРЫ.

Интерфейс Мас примечателен еще одним свойством. Без всякого принуждения его разработчики сделали его стандартным интерфейсом для всех приложений, включая подавляющее большинство написанных третьими сторонами. Таким образом, пользователь получает концептуальную когерентность на уровне интерфейса не только в программном обеспечении, поставляемом с машиной, но и во всех приложениях.

Этот подвиг дизайнеры Мас совершили, встроив интерфейс в постоянное запоминающее устройство (только для чтения), благодаря чему разработчикам проще и быстрее пользоваться им, чем создавать свои собственные зависимые и чувствительные интерфейсы. Эти естественные стимулы к единообразию преобладали достаточно широко, чтобы установить стандарт. Естественным стимулам помогла полная приверженность менеджмента и пропаганда со стороны Apple. Независимые рецензенты в журналах

о продуктах, признавая огромную ценность общей для всех приложений концептуальной целостности, также дополняли естественные стимулы, безжалостно критикуя продукты, которые не подчиняются форме.

Это превосходный пример метода, рекомендованного в главе 6, который заключается в достижении единообразия путем поощрения других к непосредственному встраиванию чьего-то кода в свои продукты, вместо того чтобы пытаться заставить их создавать свое собственное программное обеспечение по чьим-то спецификациям.

СУДЬБА WIMP: УСТАРЕВАНИЕ. Несмотря на его превосходство, я ожидаю, что через поколение интерфейс WIMP будет исторической реликвией. Указание курсором останется способом задания существительных при управлении нашими компьютерами; речь, безусловно, является правильным способом выражения глаголов. Такие инструменты, как Voice Navigator для Mac и Dragon для IBM PC, уже предоставляют эту возможность.

НЕ СОЗДАВАЙТЕ ТАК, ЧТОБЫ ПОТОМ ВЫБРАСЫВАТЬ, — ВОДОПАДНАЯ МОДЕЛЬ НЕВЕРНА!

Незабываемая картина «Галопирующей Герти» — Такомского моста — открывает главу 11, в которой радикально рекомендуется: «Планируй выкинуть первую версию; ты все равно это сделаешь!» Этот призыв радикален, потому что слишком упрощен.

Самая большая ошибка в концепции «Создавать так, чтобы потом выбросить» заключается в том, что она неявно исходит из классической последовательной, или водопадной, модели сборки ПО. Указанная модель вытекает из диаграммы Ганта поэтапного процесса, и часто рисуется так, как показано на рис. 19.1. Уинстон

Ройс (Winston Royce) улучшил последовательную модель в классической статье 1970 года, предусмотрев:

- Некоторую обратную связь от последующего этапа к предыдущему.
- Ограничение обратной связи только непосредственно предшествующим этапом, с тем чтобы ограничить затраты и задержку графика, для которых он служит поводом.

Он предшествовал «Мифическому человеку-месяцу», советуя разработчикам «строить дважды».² Глава 11 — не единственная испорченная последовательной водопадной моделью; эта ошибка проходит через всю книгу, начиная с правила планирования временного графика в главе 2. Это эмпирическое правило распределяет $\frac{1}{3}$ графика на планирование, $\frac{1}{6}$ — на написание кода, $\frac{1}{4}$ — на тестирование компонентов и $\frac{1}{4}$ — на тестирование системы.

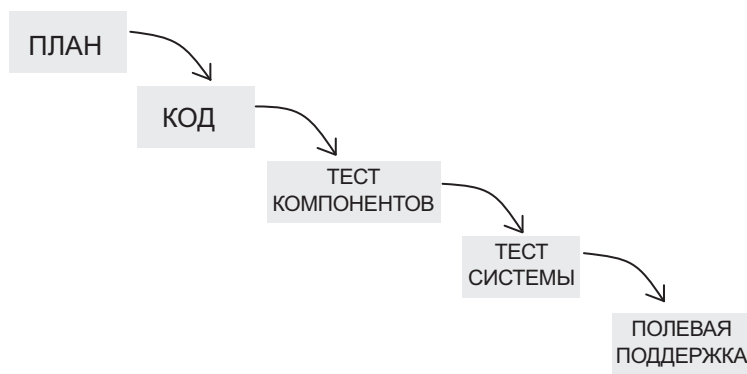


Рис. 19.1. Водопадная модель разработки ПО

Базовое заблуждение водопадной модели состоит в том, что она исходит из прохождения проектом всего процесса за *один раз*, и в том, что архитектура является превосходной и простой в использовании, дизайн имплементации является надежным, а реа-

лизация исправима в ходе тестирования. Выражаясь по-другому, водопадная модель предполагает, что все оплошности будут находиться в реализации, и, следовательно, их исправление может плавно перемежаться с тестированием компонентов и системы.

Подход «Планируй выкинуть первую версию» действительно нападает на это заблуждение. Ошибочен не диагноз, а лекарство. Далее, я действительно рекомендовал отбросить и перепланировать первую систему по частям, а не одним махом. В этом плане все в порядке, но подобному решению не удастся добраться до корня проблемы. Водопадная модель ставит тестирование системы и, следовательно, косвенным образом пользовательское тестирование в конец процесса разработки. В результате можно найти серьезные неудобства для пользователей, или неприемлемую производительность, или высокую восприимчивость к ошибке пользователя или злостному умыслу только после инвестирования в полную (завершенную) разработку. Безусловно, пристальный контроль на этапе Alpha-тестирования спецификаций призван отыскивать такие недостатки в самом начале, но ничто не может заменить привлечение настоящих пользователей.

Второе заблуждение водопадной модели состоит в том, что она предполагает сборку всей системы сразу, комбинируя куски для сквозного тестирования системы после того, как весь дизайн имплементации, большая часть написания кода и большая часть тестирования компонентов были выполнены.

Водопадная модель, которая владела умами большинства людей, занятых в проектах по созданию программного обеспечения в 1975 году, к сожалению, была закреплена в DOD-STD-2167, спецификации Министерства обороны для всего военного программного обеспечения. Это обеспечило модели долгое выживание, пока большинство вдумчивых практиков не признали ее неадекватность и от нее не отказались. К счастью, Министерство обороны с того момента прозрело.³

ДОЛЖНО БЫТЬ ДВИЖЕНИЕ ВВЕРХ ПО ТЕЧЕНИЮ. Подобно энергичному лососю на картинке в начале этой главы, опыт и идеи из каждой последующей части процесса разработки должны перескакивать вверх по течению, иногда более чем на одну стадию, и влиять на деятельность выше по течению.

Дизайн имплементации покажет, что некоторые архитектурные особенности калечат производительность; поэтому архитектура должна быть переработана. Кодирование реализации покажет, что некоторые функции раздувают потребности в пространстве; поэтому могут потребоваться изменения в архитектуре и имплементации.

Следовательно, по циклу дизайна архитектуры-имплементации вполне можно проходить два или более раза, прежде чем реализовывать что-либо в коде.

МОДЕЛЬ ИНКРЕМЕНТАЛЬНОЙ РАЗРАБОТКИ ЛУЧШЕ — ПРОГРЕССИВНОЕ УТОЧНЕНИЕ

СОЗДАНИЕ СКВОЗНОЙ СКЕЛЕТНОЙ СИСТЕМЫ. Харлан Миллс, работая в среде системы реального времени, с самого начала высказывался за то, что мы должны создать базовый опрашивающий цикл системы реального времени, с вызовами подпрограмм (*заглушками*) для всех функций (рис. 19.2), при этом сами подпрограммы должны быть пустыми. Скомпилируйте ее и протестируйте. Она будет ходить по кругу, буквально ничего не делая, но делая это правильно.⁴

Далее мы воплощаем (возможно, примитивный) входной модуль и выходной модуль. И вуаля! Работающая система, которая делает что-то, пусть и скучное. Теперь, функция за функцией, мы инкрементно создаем и добавляем модули. *На каждом этапе у нас есть работающая система.* Если мы старательны, то на каждом этапе имеем отлаженную, проверенную систему. (По мере роста систе-

мы растет и время регрессионного тестирования каждого нового модуля относительно всех предыдущих тестовых случаев.)

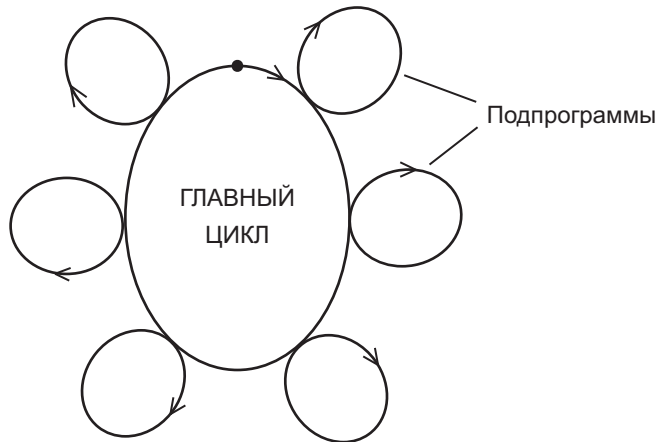


Рис. 19.2

После того как каждая функция будет работать на примитивном уровне, мы уточняем или переписываем сначала один модуль, а затем другой, постепенно *выращивая* систему. Иногда, для подстраховки, мы должны изменить исходный ведущий цикл и/или даже его модульные интерфейсы.

Так как у нас всегда есть работающая система, мы можем:

- начать тестирование со стороны пользователей очень рано;
- применить стратегию разработки по бюджету, которая полностью защищает от выхода за пределы графика или бюджета (за счет возможного функционального дефицита).

В течение примерно 22 лет я преподавал в лаборатории инженерии ПО в Университете Северной Каролины, иногда совместно с Дэвидом Парнасом. В этом курсе команды, обычно состоящие из четырех студентов, создавали за один семестр какую-то реальную

программную прикладную систему. Примерно в середине тех лет я переключился на преподавание инкрементной разработки. Я был ошеломлен электризирующим воздействием на моральный дух команды этой первой картинке на экране, этой первой работающей системы.

СЕМЕЙСТВА ПАРНАСА

Дэвид Парнас был властителем дум в инженерии ПО в течение всего этого 20-летнего периода. Все знакомы с его концепцией сокрытия информации. Менее знакомой, но не менее важной является концепция Парнаса о планировании программного продукта как *семейства* родственных продуктов.⁵ Он призывает дизайнера предвосхищать как побочные расширения, так и последующие версии продукта, а также определять их функциональные или платформенные различия, для того чтобы конструировать семейное древо родственных продуктов (рис.19.3).

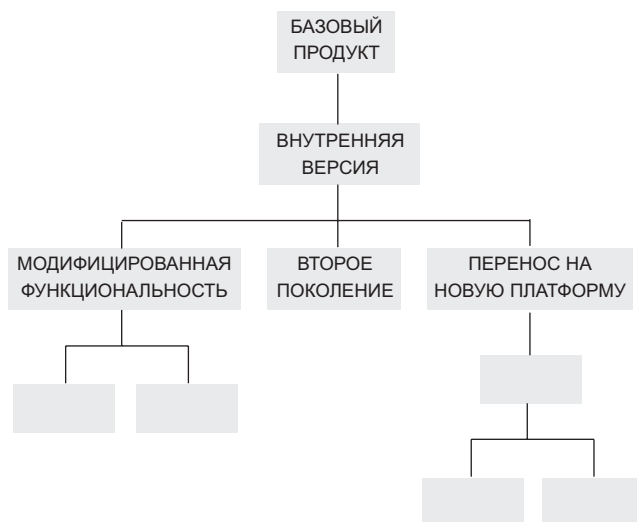


Рис. 19.3

Хитрость в планировании такого дерева заключается в том, чтобы поставить у его корня те дизайнерские решения, которые изменятся с меньшей вероятностью.

Такая стратегия дизайна максимизирует многоразовое использование модулей. Что еще более важно, одну и ту же стратегию можно расширить, включив в нее не только конечные продукты, но и преемственные промежуточные версии, созданные в рамках стратегии инкрементной разработки. Тогда продукт растет, проходя через свои промежуточные этапы, с минимальным откатом назад.

ПОДХОД «НОЧНЫХ СБОРОК» MICROSOFT

Джеймс Маккарти описал мне процесс, используемый его командой и другими сотрудниками Microsoft. Это инкрементный рост, доведенный до логического завершения. Он говорит следующее:

После того как мы выполним поставку в первый раз, мы будем поставлять более поздние версии, которые добавляют в существующий, работающий продукт функциональность. Почему первоначальный процесс разработки должен быть другим? Начиная с момента нашей первой контрольной точки (где марш к первой поставке имеет три промежуточные контрольные точки), мы собираем развивающуюся систему каждую ночь (и прогоняем тестовые случаи). Цикл сборки становится сердцебиением проекта. Каждый день одна или несколько команд программистов-тестировщиков проверяют модули с новыми функциями. После каждой сборки у нас появляется работающая система. Если сборка ломается, мы останавливаем весь процесс до тех пор, пока проблема не будет найдена и исправлена. Каждый в команде в курсе текущих дел.

Это действительно тяжело. Требуется выделение больших ресурсов, но это управляемый процесс, прослеживаемый и понятный. Он вызывает у команды доверие к себе. А доверие определяет мораль, эмоциональное состояние.

Разработчики программного обеспечения в других организациях удивлены, даже шокированы этим процессом. Один говорит: «Я взял за правило проводить сборку продукта каждую неделю, но сборки каждую ночь, думаю, это перебор с работой». И это может быть правдой. Например, Bell Northern Research осуществляют сборку своей системы из 12 миллионов строк каждую неделю.

ИНКРЕМЕНТНАЯ РАЗРАБОТКА И БЫСТРОЕ ПРОТОТИПИРОВАНИЕ

Поскольку процесс инкрементного развития допускает раннее тестирование реальными пользователями, в чем разница между ним и быстрым прототипированием? Мне кажется, что они родственны, но разделены. Можно применять один из этих подходов и при этом не применять другой.

Харел определяет прототип как

[версию программы, которая] отражает только дизайнерские решения, принятые в процессе подготовки концептуальной модели, а не решения, обусловленные вопросами имплементации.⁶

Можно построить прототип, который вовсе не является частью продукта, развивающегося в направлении поставки. Например, можно построить прототип интерфейса, который не имеет никакой реальной программной функциональности, просто конечный автомат, заставляющий его имитировать прохождение состояний. Можно даже прототипировать и тестировать интерфейсы по методике Волшебника Страны Оз, в которой спрятанный человек симулирует отклики системы. Такое прототипирование может быть очень полезным для получения ранней обратной связи с пользователем, но оно совершенно не связано с тестированием продукта, который готовится к поставке.

Схожим образом разработчики вполне могут взяться за построение вертикального среза продукта, в котором в полной мере выстроен лимитированный набор функций. Возможно, это поможет раньше пролить свет на те места, где могут скрываться змеи снижения производительности. В чем разница между сборкой в первой контрольной точке по процессу Microsoft и быстрым прототипом? В функциональности. Продукт первой контрольной точки может не иметь достаточной функциональности, чтобы представлять интерес для кого-либо; готовность продукта к поставке определяется завершенностью в предоставлении полезного набора функций и своим качеством, уверенностью в надежной работе.

ПАРНАС БЫЛ ПРАВ, А Я ОШИБАЛСЯ НАСЧЕТ СОКРЫТИЯ ИНФОРМАЦИИ

В главе 7 я противопоставляю два подхода к вопросу о том, как много каждому члену команды должно быть разрешено или предложено знать о дизайнах и коде друг друга. В проекте OS/360 мы решили, что *все* программисты должны видеть *весь* материал, то есть каждый программист должен иметь копию рабочей книги проекта, которая насчитывает более 10 000 страниц. Харлан Миллс убедительно доказывал, что «программирование должно быть публичным процессом». Если работа выставлена на всеобщее обозрение, контроль качества повышается как за счет давления со стороны коллег делать все хорошо, так и за счет коллег, фактически обнаруживающих недостатки и ошибки.

Эта точка зрения резко контрастирует с учением Дэвида Парнаса о том, что модули кода должны быть инкапсулированы с четко определенными интерфейсами и что внутренняя часть такого модуля должна быть частной собственностью его программиста, невидимой извне. Программисты являются наиболее эффективными,

если они не подвержены воздействию, а наоборот, ограждены от внутренних модулей, которые им не принадлежат.⁷

Я отверг концепцию Парнаса как «рецепт катастрофы» в главе 7. Парнас был прав, а я ошибался. Теперь я убежден, что сокрытие информации, которое сегодня часто воплощается в объектно-ориентированном программировании, является единственным способом повышения уровня дизайна программного обеспечения.

Используя другие методы, можно действительно попасть в беду. Согласно методике Миллза программисты могут получить подробности семантики интерфейсов, с которыми они работают, узнав, что находится «по ту сторону». Соккрытие этой семантики приводит к системным дефектам. С другой стороны, методика Парнаса является устойчивой в условиях изменений и более уместна в философии дизайна с учетом будущего изменения.

В главе 16 утверждается следующее:

- В прошлом бóльшая часть прогресса в продуктивности ПО происходила из устранения второстепенных трудностей, таких как неудобные машинные языки и медленный оборот пакетной обработки программ.
- Легких решений не так много.
- Радикального прогресса можно добиться, разрешив существенные сложности моделирования сложных концептуальных конструкций.

Наиболее очевидный способ добиться прогресса — признать, что программы составлены из концептуальных частей, гораздо больших, чем отдельная инструкция языка высокого уровня, — подпрограмм, модулей или классов. Если мы можем ограничить разработку и сборку так, чтобы только собирать и параметризовать такие части из предварительно построенных коллекций, то можно считать, что концептуальный уровень поднят и устранены огром-

ные объемы работы и вероятность ошибок, существующих на уровне отдельных операторов.

Данное Парнасом определение модулей с сокрытием информации было первым открытым шагом в этой критически важной программе исследований и идейным провозвестником объектно-ориентированного программирования. Он дал определение модуля как программной единицы с собственной моделью данных и собственным множеством операций. К данным модуля можно обратиться только через одну из его собственных операций. Второй шаг был вкладом нескольких мыслителей: обновление модуля Парнаса до *абстрактного типа данных*, из которого могут быть получены многие объекты. Абстрактный тип данных обеспечивает единообразный образ мышления об интерфейсах модулей и их определении, а также дисциплину доступа, которую легко обеспечить.

Третий шаг, объектно-ориентированное программирование, вводит мощную концепцию *наследования*, в соответствии с которой классы (типы данных) по умолчанию принимают в качестве значений определенные атрибуты от своих предков в иерархии классов.⁸ Большинство из того, что мы надеемся получить от объектно-ориентированного программирования, фактически вытекает из первого шага, инкапсуляции модулей, плюс идеи заранее построенных библиотек модулей или классов, которые *планируются и тестируются для переиспользования*. Многие предпочитают игнорировать тот факт, что такие модули являются не просто программами, а программными продуктами в том смысле, который обсуждается в главе 1. Некоторые тщетно надеются на значительное повторное использование модулей, не оплачивая первоначальные затраты на сборку качественных модулей — обобщенных, надежных, протестированных и задокументированных. Объектно-ориентированное программирование и повторное использование рассматриваются в главах 16 и 17.

НАСКОЛЬКО РЕАЛЕН ЧЕЛОВЕКО-МЕСЯЦ? МОДЕЛЬ И ДАННЫЕ БЁМА

В течение ряда лет были выполнены многочисленные количественные исследования производительности труда программистов и влияющих на нее факторов, особенно соотношений между обеспеченностью персоналом и графиком работ.

Наиболее значительное исследование было проведено Барри Бёмом примерно в 63 проектах по разработке программного обеспечения, в основном аэрокосмических, из которых примерно 25 были в TRW*. Его «Экономика инженерии программного обеспечения» (*Software Engineering Economics*) содержит не только результаты, но и ряд полезных моделей затрат с нарастающей сложностью. В то время как коэффициенты в моделях, безусловно, различаются для обычного коммерческого программного обеспечения и для аэрокосмического программного обеспечения, построенного по правительственным стандартам, тем не менее его модели подкреплены огромным количеством данных. Я думаю, что для следующего поколения эта книга станет классической.

Его результаты убедительно подтверждают тезис «Мифического человеко-месяца» о том, что компромисс между людьми и месяцами является далеко не линейным, что человеко-месяц является действительно мифическим как мера продуктивности. В частности, он находит, что:⁹

- Существует оптимальное по стоимости время завершения графика работ до первой поставки, $1 = 2,5 \text{ (чм)}^{1/3}$. То есть оптимальное время в месяцах выражается как кубический корень ожидаемого усилия в человеко-месяцах, цифра, полученная из

* TRW Inc. — американская корпорация, занимающаяся различными видами деятельности, в основном аэрокосмической, автомобильной и кредитной отчетностью. — *Примеч. ред.*

оценки размера и других факторов в его модели. Из этого следует оптимальная кадровая кривая.

- Кривая стоимости растет медленно по мере того, как запланированный график становится длиннее оптимального. Работа занимает все отведенное для нее время.
- Кривая стоимости растет резко по мере того, как запланированный график становится короче оптимального.
- *Практически ни один проект невозможно завершить менее чем за 3/4 от рассчитанного оптимального графика, независимо от числа привлеченных людей!* Этот цитируемый результат дает менеджеру по ПО солидный боезапас, когда высшее руководство требует принятия на себя невозможных обязательств по графику.

НАСКОЛЬКО ИСТИНЕН ЗАКОН БРУКСА? Были даже тщательные исследования с оценкой истинности закона Брукса (намеренно упрощенного), что добавление рабочей силы в запаздывающий проект по созданию ПО задержит его еще больше. Показательно это относится к Абдель-Хамиду (Abdel-Hamid) и Маднику (Madnick) в их амбициозной и ценной книге 1991 года «Динамика проектов по созданию программного обеспечения: интегрированный подход» (*Software Project Dynamics: An Integrated Approach*).¹⁰ В указанной книге развивается количественная модель динамики проекта. Глава о законе Брукса дает более подробное представление о том, что происходит при различных допущениях относительно того, какая рабочая сила добавляется и когда. В целях исследования данного вопроса авторы расширяют свою собственную тщательную модель проекта приложения среднего размера, исходя из того, что у новых людей есть кривая усвоения, и учитывая добавленную работу по коммуникации и обучению. Они приходят к выводу, что «добавление новых людей к запаздывающему проекту всегда приводит к его удорожанию, но не *всегда* к более позднему завершению» (курсив их). В частности, добавление лишней рабочей силы в начале графика

ка является гораздо более безопасным маневром, чем добавление ее позже, поскольку новые люди всегда оказывают немедленный отрицательный эффект, на компенсацию которого уходят недели.

Штутцке (Stutzke) развивает более простую модель с целью проведения аналогичного исследования, результат которого также аналогичен.¹¹ Он разрабатывает подробный анализ процесса и затрат на ассимиляцию новых работников, включая явное отвлечение их наставников от самой работы по проекту. Он тестирует свою модель относительно фактического проекта, в котором рабочая сила была успешно удвоена и первоначальный срок достигнут после отставания от графика в середине проекта. Он рассматривает альтернативы добавлению большего числа программистов, в особенности сверхурочных. Наиболее ценными являются его многочисленные практические советы о том, как следует подключать новых работников, обучать их, снабжать инструментами и т. д., чтобы минимизировать разрушительные последствия их добавления. Особенно примечателен его комментарий о том, что новые люди, пришедшие в конце проекта разработки, должны быть командными игроками, готовыми участвовать и работать в рамках процесса, а не пытаться изменить или улучшить сам процесс!

Штутцке считает, что дополнительное бремя коммуникации в более крупном проекте является эффектом второго порядка и не моделирует его. Неясно, принимают ли Абдель-Хамид и Мадник это в расчет, и если да, то как. Ни одна из моделей не учитывает тот факт, что работа должна быть перераспределена, — процесс, который я часто находил нетривиальным.

«Возмутительно упрощенная» формулировка закона Брукса становится более полезной благодаря тщательному рассмотрению надлежащих квалификаций. В целом я придерживаюсь четкой формулировки как наилучшей аппроксимации истины нулевого порядка, эмпирического правила, предупреждающего менеджеров от слепого инстинктивного исправления позднего проекта.

люди — это всё (ну почти всё)

Некоторым читателям показалось любопытным, что «Мифический человеко-месяц» уделяет большую часть эссе управленческим аспектам инженерии ПО, а не многим техническим вопросам. Этот уклон был отчасти обусловлен характером моей роли в работе над операционной системой IBM OS/360 (теперь MVS/370). Более того, он возник из убеждения, что качество людей в проекте, их организация и управление ими являются гораздо более важными факторами успеха, чем инструменты, которые они используют, или технические подходы, которые применяют.

Последующие исследования подтвердили это убеждение. Модель СОСОМО Бема находит, что качество команды является самым важным фактором ее успеха, действительно в 4 раза более мощным, чем следующий по величине фактор. Большинство научных исследований по инженерии программного обеспечения сосредоточено на инструментах. Я люблю хороший инструмент и жажду его. Тем не менее отрадно видеть продолжение исследовательских программ в отношении заботы о людях, их роста и поддержки, а также развития управления разработкой программного обеспечения.

КАДРОВОЕ ОБЕСПЕЧЕНИЕ. Крупным достижением последних лет стала книга Демарко и Листера 1987 года «Кадровое обеспечение: продуктивные проекты и команды» (*Peopleware: Productive Project sand Teams*). Ее основополагающий тезис заключается в том, что «основные проблемы нашей работы носят не столько *технологический*, сколько *социологический* характер». Она изобилует драгоценными камнями, такими как: «Функция менеджера заключается не в том, чтобы заставить людей работать, а в том, чтобы дать им возможность работать». Она имеет дело с такими привычными темами, как пространство, мебель, совместное питание команды. Демарко и Листер предоставляют реальные данные из «Программирования военных игр», которые показывают ошеломляющую корреляцию

между показателями производительности программистов в одной и той же организации, а также между характеристиками рабочего места и уровнями продуктивности и дефектности.

Пространство сотрудников с лучшей производительностью труда является более тихим, более уединенным, лучше защищенным от помех, и даже больше... Неужели для вас не важно... помогает ли тишина, пространство и уединение вашим нынешним сотрудникам делать работу лучше или же альтернативно помогает привлекать и удерживать у себя более производительных сотрудников?¹²

Я искренне рекомендую эту книгу всем моим читателям.

ПЕРЕМЕЩЕНИЕ ПРОЕКТОВ. Демарко и Листер уделяют значительное внимание *сплочению* команды — нематериальному, но жизненно важному свойству. Я думаю, что именно пренебрежение сплочением со стороны менеджмента объясняет готовность, которую я наблюдал в мультилокационных компаниях, перемещать проект из одной лаборатории в другую.

Мой опыт и наблюдения ограничены, пожалуй, половиной дюжины переездов. Успешных я не видел. Можно успешно перемещать задания. Но в каждом случае попыток переместить проекты новой команде фактически приходилось начинать сначала, несмотря на наличие хорошей документации, нескольких хорошо проработанных дизайнов и нескольких людей из отправляющей команды. Я думаю, что разрушение спаянности прежней команды приводит к выкидышу проекта, находящегося в эмбриональном состоянии, и его перезапуску.

СИЛА ОТКАЗА ОТ СИЛЫ

Если кто-то верит, что творчество исходит от людей, а не от структур или процессов, как я утверждал на многих страницах этой

книги, то главный вопрос, стоящий перед менеджером по программному обеспечению, заключается в том, как спланировать структуру и процесс так, чтобы усилить, а не подавить творчество и инициативу. К счастью, эта проблема не свойственна программным организациям, и над ней работали великие мыслители. Э. Ф. Шумахер (E. F. Schumacher) в своем классическом труде «Малое — прекрасно: экономика, в которой люди имеют значение» (*Small is Beautiful: Economics as if People Mattered*) предлагает теорию организации предприятий, чтобы максимизировать творчество и радость рабочих. В качестве своего первого принципа он выбирает «принцип субсидиарной функции» из энциклики *Quadragesima Anno* Папы Пия XI:

Будет несправедливым и в то же время тяжким злом и нарушением правильного порядка вещей возлагать на большую и высшую ассоциацию то, что могут делать меньшие и подчиненные организации. Ибо всякая общественная деятельность по самой своей природе должна оказывать помощь членам социального тела и никогда не разрушать и не поглощать их... Те, кто командует, должны быть уверены в том, что чем лучше сохраняется градуированный порядок между различными ассоциациями, соблюдая принцип субсидиарной функции, тем сильнее будет социальная власть и эффективность и тем счастливее и благополучнее будет состояние государства.¹³

Шумахер продолжает, интерпретируя следующим образом:

Принцип второстепенной функции учит нас, что центр обретет в полномочиях и эффективности, если будет бережно сохраняться свобода и ответственность низших формаций, в результате чего организация в целом будет «счастливее и более процветающей».

Как достичь такой структуры? ...Крупная организация будет состоять из множества полуавтономных единиц, которые мы можем назвать квазифирмами. Каждая из них будет обладать большой свободой, дающей максимально

возможный шанс творчеству и предпринимательству... Каждая квазифирма должна иметь как счет прибылей, так и счет убытков, а также бухгалтерский баланс.¹⁴

К числу наиболее интересных разработок в инженерии ПО относятся ранние этапы внедрения таких организационных идей на практике. Прежде всего, микрокомпьютерная революция создала новую индустрию программного обеспечения из сотен стартапов, все они начинали с малого и отличались энтузиазмом, свободой и творчеством. Сейчас отрасль меняется, так как многие малые компании приобретаются более крупными. Остается выяснить, поймут ли крупные приобретатели важность сохранения творческого потенциала малых.

Что еще примечательнее, высший менеджмент в некоторых крупных фирмах предпринимает попытки делегировать полномочия отдельным командам проекта по созданию программного обеспечения, что приблизило их к квазифирмам Шумахера по структуре и ответственности. Они удивлены и восхищены результатами.

Джим Маккарти из Microsoft описал мне свой опыт эмансипации команд:

Каждая специальная команда (feature team) (30–40 человек) владеет своим множеством функций приложения (feature set), своим расписанием и даже своим процессом того, как определять, разрабатывать и поставлять. Команда состоит из четырех или пяти специальностей, включая разработку, тестирование и документирование. Разногласия решаются внутри команды; босс не вмешивается. Не могу отдельно не подчеркнуть важность расширения полномочий, важность ответственности команды перед самой собой за успех.

Эрл Уилер (Earl Wheeler), ушедший в отставку глава отдела программного обеспечения IBM, рассказал мне о своем опыте делеги-

рования полномочий вниз, которые долгое время были централизованы в менеджменте подразделениями IBM:

Ключевым моментом [последних лет] было делегирование полномочий вниз. Это можно сравнить с волшебством! Улучшилось качество, продуктивность, моральный дух. У нас небольшие команды, без центрального управления. Команды владеют процессом, они владеют графиком, но чувствуют давление рынка. Это давление заставляет их тянуться за инструментами самостоятельно.

Беседы с отдельными членами команды, конечно же, показывают как понимание делегированного полномочия и свободы, так и несколько более консервативную оценку того, насколько на самом деле утрачен контроль. Тем не менее достигнутые делегированием успехи — это, безусловно, шаг в правильном направлении. Оно дает именно те преимущества, которые предсказывал Пий XI: центр получает реальный авторитет, делегируя полномочия, и организация в целом становится более счастливой и процветающей.

КАКИМ БЫЛ САМЫЙ БОЛЬШОЙ СЮРПРИЗ? МИЛЛИОНЫ КОМПЬЮТЕРОВ

Каждый гуру в ПО, с которым я беседовал, признается, что был застигнут врасплох микрокомпьютерной революцией и ее результатом — индустрией коммерческого программного обеспечения. Это изменение двух десятилетий после «Мифического человеко-месяца», без сомнения, является решающим. И для инженерии ПО оно имеет много последствий.

Микрокомпьютерная революция изменила принцип использования компьютеров. Более 20 лет назад Шумахер сформулировал вызов следующим образом.

Чего же мы действительно требуем от ученых и технологов? Должен ответить: нам нужны методы и оборудование, которые являются:

- дешевыми настолько, что к ним может обращаться практически каждый;
- пригодными для небольших приложений;
- соответствующими потребностям человека в творческой деятельности.¹⁵

Это именно те замечательные свойства, которые микрокомпьютерная революция принесла в компьютерную индустрию и ее пользователям и широкой публике. Средний американец теперь может позволить себе не только собственный компьютер, но и комплект программного обеспечения, который 20 лет назад стоил бы королевского содержания. Каждая из целей Шумахера заслуживает размышления; степень, в которой каждая из них была достигнута, стоит того, чтобы наслаждаться, в особенности последней. В каждой области новые средства самовыражения доступны как обычным людям, так и профессионалам.

Отчасти улучшение происходит и в других областях, как это было в создании программного обеспечения — в устранении случайных трудностей. Написание рукописей, как правило, ненамеренно осложнялось временем и стоимостью их перепечатывания с целью встроить в них изменения. На 300-страничной рукописи, бывало, приходилось делать перепечатку каждые 3–6 месяцев, а в промежутках между ними продолжать маркировку рукописи. Сложно оценить, что эти изменения сделали с потоком логики и ритмом слов. Теперь рукописи стали удивительно подвижными.¹⁶

Компьютер принес подобную подвижность во многие другие среды: рисунки, планы сооружений, механические чертежи, музыкальные композиции, фотографии, видеоряды, слайд-презентации, мультимедийные произведения — и даже в электронные таблицы. В каждом случае ручной способ производства требовал повторного

копирования громоздких неизменных частей, для того чтобы видеть изменения в контексте. Теперь в каждой из этих сред мы пользуемся теми же преимуществами, которые совместное использование времени принесло в создание программного обеспечения, — возможность мгновенно пересматривать и оценивать эффект, не теряя при этом ход своих мыслей.

Творчество также усиливается новыми и гибкими вспомогательными инструментами. Для производства текста, в качестве одного из примеров, мы теперь обеспечены орфографическими и грамматическими корректорами, стилевыми советниками, библиографическими системами и замечательной способностью видеть страницы, одновременно отформатированные в окончательный макет. Мы еще не оценили по достоинству, что значат энциклопедии мгновенного доступа или бесконечные ресурсы Всемирной паутины для импровизированного исследования писателя.

Самое главное, новая подвижность сред облегчает исследование многих радикально отличающихся альтернатив, когда творческая работа только принимает форму. Вот еще один случай, когда порядок величины в количественном параметре — в данном случае времени, необходимом для внесения изменений, — производит качественный скачок в подходе к задаче.

Инструменты для черчения позволяют архитекторам зданий разработать гораздо больше вариантов в расчете на час творческих вложений. Подключение компьютеров к синтезаторам с программным обеспечением для автоматического генерирования или воспроизведения партитур весьма облегчает захват импровизаций с клавиатуры. Цифровая обработка фотографий, как в Adobe Photoshop, позволяет в течение считанных минут провести эксперименты, для которых потребовались часы работы в фотолаборатории. Электронные таблицы позволяют легко изучить десятки альтернативных сценариев по принципу «что, если».

Наконец, благодаря повсеместному распространению персонального компьютера появились совершенно новые творческие среды. Гипертексты, предложенные Ванневаром Бушем (Vannevar Bush) в 1945 году, применимы только к компьютерам. Мультимедийные презентации и опыт были большим делом — просто несли в себе слишком много проблем — до персонального компьютера и насыщенного, дешевого программного обеспечения, доступного для него. Системы виртуального окружения, еще не ставшие дешевыми или повсеместными, станут таковыми, как и еще одной творческой средой.

МИКРОКОМПЬЮТЕРНАЯ РЕВОЛЮЦИЯ ИЗМЕНИЛА ПОДХОД К СОЗДАНИЮ ПО. Сами программные процессы 1970-х годов были изменены микропроцессорной революцией и технологическими достижениями, которые позволили это сделать. Многие частные трудности процессов разработки программного обеспечения были устранены. Быстрые индивидуальные компьютеры в настоящее время являются рутинными инструментами разработчика программного обеспечения, в результате чего понятие оборотного времени почти уже устарело. Персональный компьютер сегодня не только быстрее, чем суперкомпьютер 1960 года, он быстрее, чем рабочая станция Unix 1985 года. Все это означает, что компиляция выполняется быстро даже на самых скромных машинах, а большие объемы памяти исключают необходимость дисковой компоновки. Большие объемы памяти также делают целесообразным хранение таблиц символов в памяти с объектным кодом, поэтому высокоуровневая отладка без перекомпиляции является рутинной.

За последние 20 лет мы почти полностью пришли к совместному использованию времени в качестве методологии сборки ПО. В 1975 году совместное использование времени заменило пакетные вычисления как наиболее распространенный метод. Сеть использовалась для предоставления разработчику программного обеспечения доступа как к совместным файлам, так и к совмест-

ному мощному движку компиляции, компоновки и тестирования. Сегодня персональная рабочая станция обеспечивает вычислительный механизм, а сеть в первую очередь предоставляет совместный доступ к файлам, которые являются разрабатываемым продуктом команды. Клиент-серверные системы делают совместный доступ к регистрации, сборке и применению тестовых наборов другим и более простым процессом.

Схожие успехи были достигнуты и в пользовательских интерфейсах. Интерфейс WIMP обеспечивает гораздо более удобное редактирование текстов программ, а также текстов на английском языке. 24-строчный × 72-столбчатый экран был заменен на полностраничный или даже двухстраничный экран, поэтому программисты имеют возможность видеть гораздо больше контекста вокруг вносимых ими изменений.

ЦЕЛАЯ НОВАЯ ИНДУСТРИЯ ПО — КОММЕРЧЕСКОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Наряду с классической индустрией ПО взрывной рост претерпела еще одна. Продажи единицы продукции исчисляются сотнями тысяч, даже миллионами. Целые мощные пакеты можно приобрести по цене меньшей, чем стоимость оплаты одного рабочего дня программиста. Эти две отрасли во многом отличаются друг от друга, и они сосуществуют.

КЛАССИЧЕСКАЯ ИНДУСТРИЯ ПО. В 1975 году индустрия ПО имела несколько определяемых и несколько отличающихся компонентов, существующих и сегодня:

- Поставщики компьютеров, которые предоставляют операционные системы, компиляторы и утилиты для своих продуктов.

- Пользователи приложений, коммунальные службы, банки, страховые компании и правительственные агентства, которые создают пакеты приложений для собственного использования.
- Разработчики заказных приложений, которые заключают контракты на разработку проприетарных пакетов для пользователей. Многие из этих подрядчиков специализируются на оборонных заявках, где требования, стандарты и маркетинговые процедуры являются специфическими.
- Разработчики коммерческих пакетов, которые в то время разрабатывали в основном крупные приложения для специализированных рынков, такие как пакеты статистического анализа и САПР.

Том Демарко отмечает фрагментацию классической индустрии ПО, особенно в части пользователей приложений:

Чего я не ожидал: область разделилась на ниши. То, как вы что-то делаете, гораздо больше определяется нишей, чем использованием общих методов системного анализа, общих языков и общей методики тестирования. Ada был последним из языков общего назначения, и он стал нишевым языком.

В нише рутинных коммерческих приложений весомый вклад внесли языки четвертого поколения (4GL). Бем говорит следующее: «Наиболее успешные языки четвертого поколения являются результатом чьей-то кодификации части предметной области приложения с точки зрения опций и параметров». Наиболее распространенными из этих языков являются генераторы приложений и комбинированные пакеты, состоящие из базы данных и коммуникационного программного обеспечения с языками запросов.

Миры операционных систем срослись в единое целое. В 1975 году операционные системы изобиловали: каждый поставщик оборудования имел минимум одну собственную операционную систему

в расчете на линейку продуктов, многие имели две. И как все изменилось сегодня! Открытые системы — это лозунг, и существует всего пять значимых операционных систем, в которых люди продают пакеты приложений (в хронологическом порядке):

- Среды IBM MVS и виртуальной машины.
- Среда DEC VMS.
- Среда UNIX, в том или ином семействе.
- Среда IBM PC, будь то DOS, OS-2 или Windows.
- Среда Apple Macintosh.

ИНДУСТРИЯ КОММЕРЧЕСКОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. Для разработчика в индустрии коммерческого программного обеспечения экономика полностью отличается от экономики классической индустрии: стоимость разработки делится на большое количество единиц продукта; стоимость распространения и маркетинга достигает высоких цифр. В классической индустрии разработки приложений своими силами график и детали функциональности были предметом переговоров, стоимость разработки пересмотру не подлежала; в условиях жесткой конкуренции на открытом рынке график и функциональность уверенно доминируют над стоимостью разработки.

Как и следовало ожидать, совершенно разные экономические системы породили довольно разные культуры программирования. В классической промышленности преобладали крупные фирмы с устоявшимися стилями менеджмента и культурой труда. С другой стороны, индустрия коммерческого программного обеспечения начиналась как сотни стартапов, свободнодвигающихся и яростно сосредоточенных на результате, а не на процессе. В этом климате талант отдельного программиста всегда получал большее признание как неявное подтверждение того, что великие дизайны исходят от великих дизайнеров. Культура стартапов имеет воз-

возможность вознаграждать звездных исполнителей пропорционально их вкладу; в классической индустрии программного обеспечения корпоративная культура и зарплатная политика всегда затрудняли это. Неудивительно, что многие звезды нового поколения тяготеют к индустрии коммерческого программного обеспечения.

ПОКУПАЙ И СОЗДАВАЙ — КОММЕРЧЕСКИЕ ПАКЕТЫ В КАЧЕСТВЕ КОМПОНЕНТОВ

Радикально улучшить устойчивость программных продуктов и производительность труда при их создании можно, лишь поднявшись на один уровень и изготавливая программы из модулей или объектов. Особенно перспективным трендом является использование пакетов массового рынка в качестве платформ, на которых строятся более насыщенные и более настроенные под конкретные нужды продукты. Система слежения за грузовиками построена на основе коммерческого пакета, состоящего из базы данных и коммуникационного программного обеспечения, так же как и студенческая информационная система. Рекламные объявления в компьютерных журналах предлагают сотни стеков HyperCard и настраиваемых под собственные нужды шаблонов для Excel, десятки специальных функций на Pascal для MiniCad или функций на AutoLisp для AutoCad.

МЕТАПРОГРАММИРОВАНИЕ. Сборка стеков HyperCard, шаблонов Excel или функций MiniCad иногда называют *метапрограммированием*, конструированием нового слоя, который настраивает функцию под конкретные нужды для подмножества пользователей пакета. Концепция метапрограммирования не нова, она только возрождается и переименовывается. В начале 1960-х годов поставщики компьютеров и многие крупные поставщики менеджерских информационных систем (MIS) имели небольшие группы специалистов,

которые создавали целые языки программирования приложений из макросов на ассемблере. У поставщика MIS компании Eastman Kodak имелся внутрикорпоративный язык приложений, определенный на макроассемблере IBM 7080. Аналогично, в телекоммуникационной программе Queued Telecommunications Access Method для IBM OS/360 можно было на многих страницах кода, написанного предположительно на языке макроассемблера, не найти ни одной команды машинного уровня. Теперь блоки, предлагаемые метапрограммистам, во много раз больше, чем эти макросы. Это развитие вторичных рынков является наиболее обнадеживающим — пока мы ждали развития эффективного рынка в классах C++, незаметно вырос рынок переиспользуемых метапрограмм.

ЭТО ДЕЙСТВИТЕЛЬНО АТАКУЕТ СУЩЕСТВЕННЫЙ ПРИЗНАК. Поскольку феномен сборки на основе пакета сегодня не затрагивает среднего программиста MIS, он еще не очень заметен для дисциплины инженерии программного обеспечения. Тем не менее он будет быстро расти, потому что действительно направлен на существенный признак — конструирование концептуальных структур. Коммерческий пакет обеспечивает большой модуль функций, со сложным, но продуманным интерфейсом, и его внутренняя концептуальная структура вообще не нуждается в дизайне. Высокопроизводительные программные продукты, такие как Excel или 4th Dimension, действительно являются большими модулями, но они служат в качестве известных, задокументированных, протестированных модулей, с помощью которых можно создавать системы, настроенные под конкретные нужды. Разработчики приложений следующего уровня получают насыщенность функциональности, более короткое время разработки, протестированный компонент, более качественную документацию и радикально более низкую стоимость.

Трудность, конечно, заключается в том, что коммерческий программный пакет спланирован как автономная единица, функции и интерфейсы которой метапрограммисты не могут изменить. Что

более серьезно, разработчики коммерческих пакетов, по-видимому, имеют мало стимулов для того, чтобы делать свои продукты подходящими в качестве модулей в более крупной системе. Я думаю, что это восприятие является неправильным, что существует неосвоенный рынок в предоставлении пакетов, предназначенных для того, чтобы способствовать повышению полезности метапрограммистов.

ТАК ЧТО ЖЕ НУЖНО? Мы можем выявить четыре уровня пользователей коммерческих пакетов:

- Пользователь как есть, который работает с приложением прямым способом, довольствуясь функциями и интерфейсом, предоставляемыми дизайнерами.
- Метапрограммист, который создает шаблоны или функции поверх отдельного приложения, используя предоставленный интерфейс, главным образом с целью экономии работы для конечного пользователя.
- Составитель внешних функций, который вручную кодирует функции, добавляемые в приложение. Они представляют собой принципиально новые примитивы языка приложений, которые вызывают отдельные модули кода, написанные на языке общего назначения. Здесь требуется возможность взаимодействовать с приложением через перехват команд, функции обратного вызова и переопределение функций.
- Метапрограммист, который использует одно или, в особенности, несколько приложений в качестве компонентов в более крупной системе. Это тот пользователь, чьи потребности сегодня плохо удовлетворяются. Благодаря такому использованию можно ожидать существенного повышения эффективности при разработке новых приложений.

Для конечного пользователя коммерческое приложение нуждается в дополнительном документированном интерфейсе — интерфейсе метапрограммирования (MPI). Для этого нужно несколько ус-

ловий. Во-первых, метапрограмма должна управлять ансамблем приложений, тогда как в обычной ситуации каждое приложение исходит из того, что оно само контролирует себя. Ансамбль должен управлять пользовательским интерфейсом, тогда как обычно предполагается, что за это отвечает приложение. Ансамбль должен иметь возможность вызывать любую функцию приложения, как если бы команда управления исходила от пользователя. Выходные данные приложения должны передаваться ему, а не на экран, причем в виде логических блоков подходящих типов данных, а не текстовой строки, которую нужно отобразить. Некоторые приложения, такие как FoxPro, имеют червоточины (wormhole), через которые можно передавать команду в виде строки, но информация, которая приходит обратно, является скудной и неструктурированной. Такая червоточина — это ситуативное (внесистемное, *ad hoc*) исправление для потребности, которая требует общего, спланированного решения.

Очень важно иметь язык сценариев для управления взаимодействиями внутри ансамбля приложений. UNIX впервые предоставил такую функциональность, со своими каналами (pipes) и стандартным форматом текстового файла ASCII. Сегодня Apple Script является довольно хорошим примером.

СОСТОЯНИЕ И БУДУЩЕЕ ИНЖЕНЕРИИ ПО

Однажды я попросил Джима Феррелла (Jim Ferrell), заведующего кафедрой химической инженерии в Университете штата Северная Каролина, рассказать об истории химической инженерии как отдельного направления химии. В этой связи он дал замечательный импровизированный часовой отчет, начиная с того, что со времен античности существовало много разных производственных процессов для многих продуктов, от стали до хлеба и духов. Он рассказал, как профессор Артур Д. Литтл (Arthur D. Little) в 1918 году основал

в MIT факультет прикладной химии для исследования, разработки и обучения общим фундаментальным технологиям всех процессов. Сначала появились эмпирические правила, затем эмпирические номограммы, затем формулы для сборки отдельных компонентов, затем математические модели теплопереноса, массопереноса, переноса импульса в отдельных сосудах.

По мере того как Феррелл рассказывал, я был поражен многочисленными параллелями между развитием химической инженерии и программной инженерии почти ровно 50 лет спустя. Парнас упрекает меня за то, что я вообще пишу *об инженерии программного обеспечения*. Он противопоставляет программную дисциплину электротехнике и считает самонадеянным называть то, что мы делаем, инженерией. Возможно, он прав в том, что эта область никогда не превратится в инженерную дисциплину с такой точной и всеобъемлющей математической базой, как электротехника. В конце концов, программная инженерия, как и химическая инженерия, занимается нелинейными задачами вертикального масштабирования процессов промышленного масштаба, и, как и промышленная инженерия, ее постоянно сбивает с толку сложность человеческого поведения.

Тем не менее ход и сроки развития химической инженерии приводят меня к мысли, что программная инженерия в возрасте 27 лет может быть не безнадежной, а просто незрелой, как это было с химией в 1945 году. Только после Второй мировой войны инженеры-химики действительно обратили внимание на поведение замкнутых взаимосвязанных систем непрерывного потока.

Отличительные вопросы инженерии программного обеспечения сегодня именно те, которые изложены в главе 1:

- Как спланировать и собрать множество программ в *систему*.
- Как спланировать и создать программу или систему, являющуюся надежным, протестированным, задокументированным, поддерживаемым *продуктом*.

- Как сохранить интеллектуальный контроль в условиях большой *сложности*.

Смоляная яма инженерии ПО будет оставаться вязкой еще долгое время. Можно ожидать, что человечество продолжит попытки разработки систем, находящихся в пределах или за пределами нашей досягаемости; а программные системы, возможно, так и останутся наиболее сложными человеческими творениями. Это непростое ремесло потребует от нас непрерывно развивать эту дисциплину, учиться создавать из более крупных блоков, наилучшим образом использовать новые инструменты, старательно осваивать опробованные методы управления инженерией, щедро использовать здравый смысл и смиренно сознавать свою подверженность ошибкам и ограниченность возможностей.

ЭПИЛОГ

ПЯТЬДЕСЯТ ЛЕТ УДИВЛЕНИЯ, ВООДУШЕВЛЕНИЯ И РАДОСТИ

До сих пор живы в моей памяти удивление и восторг, с которыми я — тогда мне было 13 лет — читал отчет от 7 августа 1944 года, посвященный компьютеру Mark I, архитектором которого был Говард Айкен (Howard Aiken), а проектировщиками — инженеры Клер Лейк (Clair D. Lake), Бенджамин Дурфи (B. M. Durfee) и Фрэнсис Гамильтон (F. E. Hamilton). Не менее удивительным было чтение статьи Ванневара Буша «Что мы можем подумать» (*That We May Think*) в журнале *Atlantic Monthly* за апрель 1945 года, в которой он предложил организовать знание в виде большой гипертекстовой сети и дать пользователям машины с возможностью как следовать по существующим тропам, так и прокладывать новые тропы ассоциаций.

Моя страсть к компьютерам получила еще один сильный импульс в 1952 году, когда летняя работа в *IBM* в Эндикотте, штат Нью-Йорк, дала мне практический опыт программирования IBM 604 и формальные знания по программированию IBM 701, первой машины с хранимой программой. Аспирантура под руководством Айкена и Айверсона в Гарварде сделала мою мечту о карьере реаль-

ностью, и меня зацепило на всю жизнь. Лишь малой части человеческого рода Бог дает привилегию зарабатывать себе на хлеб, делая то, к чему человек охотно стремился бы бесплатно, ради страсти. И я весьма благодарен.

Для человека, влюбленного в компьютеры, трудно было бы придумать иное время, когда так радостно было жить. От механизмов до вакуумных ламп, транзисторов и интегральных схем шло бурное развитие технологий. Первый компьютер, на котором я работал, только что окончив Гарвард, был суперкомпьютер IBM 7030 Stretch. Stretch был самым быстрым компьютером в мире с 1961 по 1964 год; было выпущено девять его копий. Мой Macintosh Powerbook сегодня не только быстрее, с большей памятью и большим диском, но и в тысячу раз дешевле. (В пять тысяч раз дешевле в долларах с постоянной покупательной способностью.) Мы стали свидетелями, по очереди, компьютерной революции, электронно-вычислительной революции, мини-компьютерной революции и микрокомпьютерной революции, каждая из которых приносила на порядок больше компьютеров.

Интеллектуальная дисциплина, связанная с компьютером, получила взрывное развитие, как и технология. Когда я был аспирантом в середине 1950-х годов, я мог читать все журналы и материалы конференций; я мог оставаться в курсе всего в индустрии. В моей сегодняшней интеллектуальной жизни можно наблюдать, как я с сожалением прощаюсь с субдисциплинарными интересами, один за другим, поскольку мое портфолио постоянно заполняется, выливаясь за пределы возможностей. Слишком много интересов, слишком много захватывающих возможностей для усвоения, исследований и размышлений. Какое изумительное недоразумение! Мало того что конца этому не видно, так еще и темп не замедляется. У нас много будущих радостей.

ЗАМЕТКИ И ССЫЛКИ

ГЛАВА 1

1. Ершов считает это не только горестью, но отчасти и радостью. A. P. Ershov. Aesthetics and the human factor in programming // CACM. 1972. Vol. 15, N 7. July. P. 501-505

ГЛАВА 2

1. V. A. Vyssotsky из Bell Telephone Laboratories считает, что крупный проект может выдержать наращивание рабочей силы на 30 % в год. Наращивание напрягает и даже тормозит эволюцию существенной неформальной структуры и ее коммуникационных маршрутов, обсуждаемых в главе 7.
2. F. J. Corbato из MIT указывает на то, что длительный проект должен предвосхищать оборот персонала в размере 20 % в год, и он должен быть как технически обучен, так и интегрирован в формальную структуру.
3. C. Portman из International Computers Limited говорит: «Когда было отмечено, что все работает, все интегрировано, у вас есть еще четыре месяца работы». Несколько других наборов деления графика приведено в Wolverson, R. W., «The cost of developing large-scale software», IEEE Trans, on Computers, C-23, 6 (June, 1974) pp. 615–636.

4. Рисунки 2.5–2.8 выполнены Джерри Огдиным (Jerry Ogdin), который, цитируя мой пример из более ранней публикации этой главы, значительно улучшил рисунок. Ogdin, J. L., «The Mongolian hordes versus super programmer», *Infosystems* (Dec., 1972), pp. 20–23.

ГЛАВА 3

1. Sackman, H., W. J. Erikson, and E. E. Grant, «Exploratory experimental studies comparing online and offline programming performance», *CACM*, 11, 1 (Jan., 1968), pp. 3–11.
2. Mills, H., «Chief programmer teams, principles, and procedures», IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Md., 1971.
3. Baker, F. T., «Chief programmer team management of production programming», *IBM Sys. J.*, 11, 1 (1972).

ГЛАВА 4

1. Eschapasse, M., *Reims Cathedral*, Caisse Nationale des Monuments Historiques, Paris, 1967.
2. Brooks, F. P., «Architectural philosophy», in W. Buchholz (ed.), *Planning a Computer System*. New York: McGraw-Hill, 1962.
3. Blaauw, G. A., «Hardware requirements for the fourth generation», in F. Gruenberger (ed.), *Fourth Generation Computers*. Englewood Cliffs, N.J.: Prentice-Hall, 1970.
4. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, Chapter 5.
5. Glegg, G. L., *The Design of Design*. Cambridge: Cambridge Univ. Press, 1969, он говорит следующее: «На первый взгляд идея о том, что какие-то правила или принципы накладываются на творческий ум, скорее мешает, чем помогает, но это совершенно неверно на практике. Дисциплинированное мышление фокусирует вдохновение, а не ослепляет его».

6. Conway, R. W., «The PL/C Compiler», Proceedings of a Conf. on Definition and Implementation of Universal Programming Languages. Stuttgart, 1970.
7. Хорошее обсуждение необходимости технологии программирования см. С. Н. Reynolds, «What's wrong with computer programming management?» in G. F. Weinworm (ed.), On the Management of Computer Programming. Philadelphia: Auerbach, 1971, pp. 35–42.

ГЛАВА 5

1. Strachey, C., «Review of Planning a Computer System», Comp., 5, 2 (July, 1962), pp. 152–153.
2. Это относится только к управляющим программам. Некоторые компиляторные команды разработчиков в рамках усилия по сборке OS/360 создавали свои третьи или четвертые системы, и совершенство их продуктов показывает это.
3. Shell, D. L., «The Share 709 system: a cooperative effort»; Greenwald, I. D., and M. Kane, «The Share 709 system: programming and modification»; Boehm, E. M., and T. B. Steel, Jr., «The Share 709 system: machine implementation of symbolic programming»; all in /ACM, 6, 2 (April, 1959), pp.123–140.

ГЛАВА 6

1. Neustadt, R. E., Presidential Power. New York: Wiley, 1960, Chapter 2.
2. Backus, J. W., «The syntax and semantics of the proposed international algebraic language». Proc. Intl. Con/. Inf. Proc. UNESCO, Paris, 1959, published by R. Oldenbourg, Munich, and Butterworth, London. Кроме нее целая коллекция работ на эту тему содержится в Т. В. Steel, Jr. (ed.), Formal Language Description Languages for Computer Programming. Amsterdam: North Holland, (1966).
3. Lucas, P., and K. Walk, «On the formal description of PL/I», Annual Review in Automatic Programming Language. New York: Wiley, 1962, Chapter 2, p. 2.

4. Iverson, K. E., A Programming Language. New York: Wiley, 1962, Chapter 2.
5. Falkoff, A. D., K. E. Iverson, E. H. Sussenguth, «A formal description of System/360», ZBM Systems Journal, 3, 3 (1964), pp. 198-261.
6. Bell, C. G., and A. Newell, Computer Structures. New York: McGraw-Hill, 1970, pp. 120–136, 517–541.
7. Bell, C. G., частное общение.

ГЛАВА 7

1. Parnas, D. L., «Information distribution aspects of design methodology», Carnegie-Mellon Univ., Dept. of Computer Science Technical Report, February, 1971.
2. Copyright 1939, 1940 Street & Smith Publications, Copyright 1950, 1967 by Robert A. Heinlein. Published by arrangement with Spectrum Literary Agency.

ГЛАВА 8

1. Sackman, H., W.J. Erikson, and E. E. Grant, «Exploratory experimentation studies comparing online and offline programming performance», CACM, 11, 1 (Jan., 1968), pp. 3–11.
2. Nanus, B., and L. Farr, «Some cost contributors to largescale programs», AFIPS Proc. SJCC, 25 (Spring, 1964), pp. 239–248.
3. Weinwurm, G. F., «Research in the management of computer programming», Report SP-2059, System Development Corp., Santa Monica, 1965.
4. Morin, L. H., «Estimation of resources for computer programming projects», M. S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Portman, C., частное общение.
6. Неопубликованное исследование 1964 года. E. F. Bardain показывает, что программисты реализуют 27 % продуктивного времени. (Цити-

- руется по D. B. Mayer and A. W. Stalnaker, «Selection and evaluation of computer personnel. Proc. 23rd ACM. Con., 1968, p. 661.)
7. Aron, J., частное общение.
 8. Доклад, представленный на панельной сессии и не включенный в материалы AFIPS Proceedings.
 9. Wolverton, R. W., «The cost of developing large-scale software», IEEE Trans, on Computers, C-23, 6 (June, 1974) pp. 615–636. Этот важный недавний документ содержит данные по многим вопросам этой главы, а также подтверждает выводы о продуктивности.
 10. Corbato, F. J., «Sensitive issues in the design of multi-use systems», lecture at the opening of the Honeywell EDP Technology Center, 1968.
 11. W. M. Taliaffero also reports a constant productivity of 2400 statements/year in assembler, Fortran, and Cobol. See «Modularity. The key to system growth potential», Software, I, 3 (July 1971) pp. 245–257.
 12. E. A. Nelson's System Development Corp. Report TM-3225, Management Handbook for the Estimation of Computer Programming Costs, показывает повышение продуктивности в три раза для языка высокого уровня (стр. 66–67), хотя его стандартные отклонения имеют широкие значения.

ГЛАВА 9

1. Brooks, F. P. and K. E. Iverson, Automatic Data Processing, System/360 Edition. New York: Wiley, 1969, Chapter 6.
2. Knuth, D. E., The Art of Computer Programming, Vols. 1-3. Reading, Mass.: Addison-Wesley, 1968.

ГЛАВА 10

1. Conway, M. E., «How do committees invent?», Datamation, 14, 4 (April, 1968), pp. 28–31.

ГЛАВА 11

1. Speech at Oglethorpe University, May 22, 1932.
2. Яркий отчет об опыте Multics на двух преемственных системах содержится в F. J. Corbatd, J. H. Saltzer, and C. T. Clingen, «Multics — the first seven years», AFIPS Proc S/CC, 40 (1972), pp. 571–583.
3. Cosgrove, J., «Needed: a new planning framework», Datamation, 17, 23 (Dec., 1971), pp. 37–39.
4. Вопрос изменения дизайна является сложным, и я здесь слишком упрощаю. См. J. H. Saltzer, «Evolutionary design of complex systems», in D. Eckman (ed.), Systems: Research and Design. New York: Wiley, 1961. Однако когда все сказано и сделано, я все еще выступаю за сборку пилотной системы, отказ от которой запланирован.
5. Campbell, E., «Report to the AEC Computer Information Meeting», December, 1970. Это явление также обсуждается в J. L. Ogdin, «Designing reliable software», Datamation, 18, 7 (July, 1972), pp. 71–78. Мои опытные друзья, кажется, довольно равномерно разделились относительно того, идет или нет кривая снова вниз.
6. Lehman, M., and L. Belady, «Programming system dynamics», given at the ACM SIGOPS Third Symposium on Operating System Principles, October, 1971.
7. Lewis, C. S., Mere Christianity. New York: Macmillan, 1960, p. 54.

ГЛАВА 12

1. См. также: J. W. Pomeroy, «A guide to programming tools and techniques», IBM Sys. J., 11, 3 (1972), pp. 234–254.
2. Landy, B., and R. M. Needham, «Software engineering techniques used in the development of the Cambridge Multiple-Access System», Software, 1, 2 (April, 1971), pp. 167–173.
3. Corbatd, F. J., «PL/I as a tool for system programming», Datamation, 15, 5 (May, 1969), pp. 68–76.

4. Hopkins, M., «Problems of PL/I for system programming», IBM Research Report RC 3489, Yorktown Heights, N.Y., August 5, 1971.
5. Corbatd, F. J., J. H. Saltzer, and C. T. Clingen, «Multics — the first seven years», AFIPS Proc SJCC, 40 (1972), pp. 571–582. «Только в половине областей, которые были написаны на PL/1, были сделаны отступления на машинном языке по причинам выжимания максимальной производительности. В нескольких программах, которые первоначально были на машинном языке, был сделан переход на PL/I для повышения их уровня сопровождаемости».
6. Цитируя статью Корбатто, приведенную в ссылке 3: «PL/I is here now and the alternatives are still untested». Но см. совершенно противоположное, хорошо задокументированное мнение в Henricksen, J. O. and R. E. Merwin, «Programming language efficiency in real-time software systems», AFIPS Proc SJCC, 40 (1972) pp. 155–161.
7. Не все согласны. В частном общении Харлан Миллс говорит следующее: «Опыт начинает говорить мне, что в производственном программировании человек, которого нужно поставить на терминал, — это секретарь. Идея состоит в том, чтобы сделать программирование более публичной практикой, под общим наблюдением многих членов команды, а не частным ремеслом».
8. Harr, J., «Programming Experience for the Number 1 Electronic Switching System», paper given at the 1969 SJCC.

ГЛАВА 13

1. Vyssotsky, V. A., «Common sense in designing testable software», lecture at The Computer Program Test Methods Symposium, Chapel Hill, N.C., 1972. Most of Vyssotsky's lecture is contained in Hetzel, W. C. (ed.), Program Test Methods. Englewood Cliffs, N.J.: Prentice-Hall, 1972, pp. 41–47.
2. Wirth, N., «Program development by stepwise refinement», CACM 14, 4 (April, 1971), pp. 221–227. См. также: Mills, H. «Top-down programming in large systems», in R. Rustin (ed.) Debugging Techniques in Large Systems. Englewood Cliffs, N.J.: Prentice-Hall, 1971, pp. 41–55 and Baker, F. T., «System quality through structured programming», AFIPS Proc FJCC, 41-1 (1972), pp. 339–343.

3. Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. London and New York: Academic Press, 1972. Этот том содержит самую полную трактовку. См. также письмо Дейкстры: «GOTO statement considered harmful», CACM, II, 3 (March, 1968), pp. 147–148.
4. Bohm, C., and A. Jacopini, «Flow diagrams, Turing machines, and languages with only two formation rules», CACM, 9, 5 (May, 1966), pp. 366–371.
5. Codd, E. F., E. S. Lowry, E. McDonough, and C. A. Scalzi, «Multiprogramming STRETCH: Feasibility considerations», CACM, 2, 11 (Nov., 1959), pp. 13–17.
6. Strachey, C., «Time sharing in large fast computers», Proc. Int. Con/, on Info. Processing, UNESCO (June, 1959), pp. 336–341. См. также замечания Кодда (Codd) на с. 341, где он сообщил о прогрессе в работе, подобной той, которая была предложена в документе Стрейчи.
7. Corbato, F. J., M. Merwin-Daggett, R. C. Daley, «An experimental time-sharing system», AFIPS Proc. SJCC, 2, (1962), pp. 335–344. Reprinted in S. Rosen, *Programming Systems and Languages*. New York: McGraw-Hill, 1967, pp. 683–698.
8. Gold, M. M., «A methodology for evaluating time-shared computer system usage», Ph.D. dissertation, Carnegie-Mellon University, 1967, p. 100.
9. Gruenberger, F., «Program testing and validating», Datamation, 14, 7, (July, 1968), pp. 39–47.
10. Ralston, A., *Introduction to Programming and Computer Science*. New York: McGraw-Hill, 1971, pp. 237–244.
11. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, pp. 296–299.
12. Хорошая трактовка разработки спецификаций и сборки и тестирования систем дана в F. M. Trapnell, «A systematic approach to the development of system programs», AFIPS Proc SJCC, 34 (1969) pp. 411–418.
13. Для системы реального времени потребуется симулятор среды. См., например, в M. G. Ginzberg, «Notes on testing real-time system programs», IBM Sys. J., 4, 1 (1965), pp. 58–72.
14. Lehman, M., and L. Belady, «Programming system dynamics», given at the ACM SIGOPS Third Symposium on Operating System Principles, October, 1971.

ГЛАВА 14

1. См. С. Н. Reynolds, «What's wrong with computer programming management?» в G. F. Weinwurm (ed.), *On the Management of Computer Programming*. Philadelphia: Auerbach, 1971, pp. 35–42.
2. King, W. R., and T. A. Wilson, «Subjective time estimates in critical path planning-a preliminary analysis», *Mgt. Sci.*, 13, 5 (Jan., 1967), pp. 307–320, and sequel, W. R. King, D. M. Witterrongel, K. D. Hezel, «On the analysis of critical path time estimating behavior», *Mgt. Sci.*, 14, 1 (Sept., 1967), pp. 79–84.
3. Более полное обсуждение см. в Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*, New York: Wiley, 1969, pp. 428–430.
4. Частное общение.

ГЛАВА 15

1. Goldstine, H. H., and J. von Neumann, «Planning and coding problems for an electronic computing instrument», Part II, Vol. 1, report prepared for the U.S. Army Ordinance Department, 1947; reprinted in J. van Neumann, *Collected Works*, A. H. Taub (ed.), Vol. v, New York: McMillan, pp. 80–151.
2. Частное общение, 1957. Указанный аргумент опубликован в Iverson, K. E., «The Use of APL in Teaching», Yorktown, N.Y.: IBM Corp., 1969.
3. Еще один список методов для PL/1 дается в A. B. Walter and M. Bohl, «From better to best-tips for good programming», *Software Age*, 3, 11 (Nov., 1969), pp. 46–50.

Те же методы могут быть использованы в Algol и даже Fortran. У Д. Э. Ланга (D. E. Lang) из Университета Колорадо есть программа форматирования Fortran под названием STYLE, которая достигает такого результата. См. также D. D. McCracken and G. M. Weinberg, «How to write a readable FORTRAN program», *Datamation*, 18, 10 (Oct., 1972), pp. 73–77.

ГЛАВА 16

1. Эссе, озаглавленное «Серебряной пули нет», взято из Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference, edited by H.-J. Kugler (1986), pp. 1069–76. Reprinted with the kind permission of IFIP and Elsevier Science B. V., Amsterdam, The Netherlands.
2. Parnas, D. L., «Designing software for ease of extension and contraction», IEEE Trans. on SE, 5, 2 (March, 1979), pp. 128–138.
3. Booch, G., «Object-oriented design», in Software Engineering with Ada. Menlo Park, Calif.: Benjamin/Cummings, 1983.
4. Mostow, J., ed., Special Issue on Artificial Intelligence and Software Engineering, IEEE Trans. on SE, 11, 11 (Nov., 1985).
5. Parnas, D. L., «Software aspects of strategic defense systems», Communications of the ACM, 28, 12 (Dec., 1985), pp. 1326–1335. Also in American Scientist, 73, 5 (Sept.-Oct., 1985), pp. 432–440.
6. Balzer, R., «A IS-year perspective on automatic programming», in Mostow, op. cit.
7. Mostow, op. cit.
8. Parnas, 1985, op. cit.
9. Raeder, G., «A survey of current graphical programming techniques», in R. B. Grafton and T. Ichikawa, eds., Special Issue on Visual Programming, Computer, 18, 8 (Aug., 1985), pp. 11–25.
10. Эта тема обсуждается в главе 15 данной книги.
11. Mills, H. D., «Top-down programming in large systems», Debugging Techniques in Large Systems, R. Rustin, ed., Englewood Cliffs, N.J., Prentice-Hall, 1971.
12. Boehm, B. W., «A spiral model of software development and enhancement», Computer, 20, 5 (May, 1985), pp. 43–57.

ГЛАВА 17

1. Материал, приведенный без цитирования, взят из частных бесед.
2. Brooks, F. P., «No silver bullet-essence and accidents of software engineering», in *Information Processing 86*, H. J. Kugler, ed. Amsterdam: Elsevier Science (North Holland), 1986, pp. 1069–1076.
3. Brooks, F. P., «No silver bullet-essence and accidents of software engineering», *Computer* 20, 4 (April, 1987), pp. 10–19.
4. Несколько писем и ответы на них появились в июльском номере журнала *Computer* за 1987 год. Особенно приятно отметить, что в то время как статья «Серебряной пули нет» не получила никаких наград, обзор Брюса М. Сквирски (Bruce M. Skwiersky) был выбран как лучший обзор, опубликованный в *Computing Reviews* в 1988 году. Е. А. Вайс (E. A. Weiss), «Editorial», *Computing Reviews* (June, 1989), pp. 283–284, объявляет награду и перепечатывает комментарий Брюса М. Сквирски. В обзоре есть одна существенная ошибка: вместо «шестикратный» должен быть « 10^6 ».
5. «Согласно Аристотелю и схоластической философии, случайность — это признак, который не принадлежит вещи по праву ее сущностной или субстанциальной природы, но возникает в ней как следствие других причин». Новый международный словарь английского языка Webster's New International Dictionary of the English Language, 2d ed., Springfield, Mass.: G. C. Merriam, 1960.
6. Sayers, Dorothy L., *The Mind of the Maker*. New York: Harcourt, Brace, 1941.
7. Glass, R. L., and S. A. Conger, «Research software tasks Intellectual or clerical?», *Information and Management*, 23, 4(1992). Авторы сообщают об измерении требований к программному обеспечению, где около 80 % являются интеллектуальными и 20 % — бюрократическими. Филстад (Fjelstadt) и Хэмлин (Hamlen), 1979, получают, по существу, те же результаты для сопровождения прикладного программного обеспечения. Я не знаю ни одной успешной попытки измерить эту долю для всей сквозной работы целиком.
8. Herzberg, F., B. Mausner, and B. B. Sayderman. *The Motivation to Work*, 2nd ed. London: Wiley, 1959.
9. Cox, B. J., «There is a silver bullet», *Byte* (Oct., 1990), pp. 209–218.

10. Harel, D., «Biting the silver bullet: Toward a brighter future for system development», *Computer* (Jan., 1992), pp. 8–20.
11. Parnas, D. L., «Software aspects of strategic defense systems», *Communications of the ACM*, 28, 12 (Dec., 1985), pp. 1326–1335.
12. Turski, W. M., «And no philosophers' stone, either», in *Information Processing 86*, H. J. Kugler, ed. Amsterdam: Elsevier Science (North Holland), 1986, pp. 1077–1080.
13. Glass, R. L., and S. A. Conger, «Research Software Tasks: Intellectual or Clerical?» *Information and Management*, 23, 4 (1992), pp. 183–192.
14. Review of Electronic Digital Computers, *Proceedings of a Joint AIEE-IRE Computer Conference* (Philadelphia, Dec. 10–12, 1951). New York: American Institute of Electrical Engineers, pp. 13–20.
15. Там же, pp. 36, 68, 71, 97.
16. *Proceedings of the Eastern Joint Computer Conference*, (Washington, Dec. 8–10, 1953). New York: Institute of Electrical Engineers, pp. 45–47.
17. *Proceedings of the 1955 Western Joint Computer Conference* (Los Angeles, March 1–3, 1955). New York: Institute of Electrical Engineers.
18. Everett, R. R., C. A. Zraket, and H. D. Bennington, «SAGE-A data processing system for air defense», *Proceedings of the Eastern Joint Computer Conference*, (Washington, Dec. 11–13, 1957). New York: Institute of Electrical Engineers.
19. Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, «Statemate: A working environment for the development of complex reactive systems», *IEEE Trans. on SE*, 16, 4 (1990), pp. 403–444.
20. Jones, C., *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Prentice-Hall, 1994. p. 619.
21. Coqui, H., «Corporate survival: The software dimension», *Focus '89*, Cannes, 1989.
22. Coggins, James M., «Designing C++ libraries», *C++ Journal*, 1, 1 (June, 1990), pp. 25–32.
23. Время — будущее; мне не известно ни одного такого результата, о котором бы сообщалось для пятого использования.
24. Jones, op. cit., p 604.

25. Huang, Weigiao, «Industrializing software production», Proceedings ACM 1988 Computer Science Conference, Atlanta, 1988. Боюсь отсутствия личного роста при таком раскладе.
26. Весь сентябрь 1994 года выпуск IEEE Software посвящен повторному использованию.
27. Jones, op. cit., p. 323.
28. Jones, op. cit., p. 329.
29. Yourdon, E., Decline and Fall of the American Programmer. Englewood Cliffs, N.J.: Yourdon Press, 1992, p. 221.
30. Glass, R. L., «Glass» (column), System Development, (January, 1988), pp. 4–5.

ГЛАВА 18

1. Boehm, B. W., Software Engineering Economics, Englewood Cliffs, N.J.: Prentice-Hall, 1981, pp. 81–84.
2. McCarthy, J., «21 Rules for Delivering Great Software on Time», Software World USA Conference, Washington (Sept., 1994)

ГЛАВА 19

1. Материал, приведенный без цитирования, взят из частных бесед.
2. По этому болезненному вопросу см. также: Niklaus Wirth «A plea for lean software», Computer, 28, 2 (Feb., 1995), pp. 64–68.
3. Coleman, D., 1994, «Word 6.0 packs in features; update slowed by baggage», MacWeek, 8, 38 (Sept. 26, 1994), p. 1.
4. Были опубликованы многочисленные обзоры частот команд машинного языка и языка программирования после того, как было опубликовано развертывание. Например, см.: J. Hennessy and D. Patterson, Computer Architecture. Эти частотные данные очень полезны для создания последующих продуктов, хотя они никогда в точности не соблюдаются. Я не знаю ни одной опубликованной оценки частоты,

подготовленной до создания дизайна продукта, а тем более сравнения априорных оценок и апостериорных данных. Кен Брукс полагает, что доски объявлений в Интернете предоставляют теперь дешевый способ запросить данные у предполагаемых пользователей нового продукта, даже несмотря на то что отвечают только желающие.

5. Conklin, J., and M. Begeman, «gIBIS: A Hypertext Tool for Exploratory Policy Discussion», *ACM Transactions on Office Information Systems*, Oct. 1988, pp. 303–331.
6. Englebart, D., and W. English, 1/ A research center for augmenting human intellect», *AFIPS Conference Proceedings*, Fall Joint Computer Conference, San Francisco (Dec. 9-11, 1968), pp. 395–410.
7. Apple Computer, Inc., *Macintosh Human Interface Guidelines*, Reading, Mass.: Addison-Wesley, 1992.
8. Похоже, что настольная шина Apple может обрабатывать две мыши программно, но операционная система не предоставляет такой функции.
9. Royce, W. W., 1970. «Managing the development of large software systems: Concepts and techniques», *Proceedings, WESCON* (Aug., 1970), reprinted in the *ICSE 9 Proceedings*. Ни Ройс, ни другие не верили, что можно пройти через программный процесс без ревизии более ранних документов; указанная модель была выдвинута как идеал и концептуальная помощь. См. D. L. Parnas and P. C. Clements, «A rational design process: How and why to fake it», *IEEE Transactions on Software Engineering*, SE-12, 2 (Feb., 1986), pp. 251–257.
10. Крупная переработка DOD-STD-2167 произвела DOD-STD-2167A (1988), который позволяет, но не требует более поздних моделей, таких как спиральная модель. К сожалению, MILSPECS, на которые ссылается 2167A, и иллюстративные примеры, которые он использует, по-прежнему ориентированы на водопадную модель, поэтому большинство закупок продолжают использовать водопадную модель, как сообщает Бём (Boehm). Целевая группа Научного совета Министерства обороны США под руководством Ларри Друффела и Джорджа Хейлмейера в своем докладе 1994 года «Отчет целевой группы DSB о приобретении оборонного программного обеспечения на коммерческой основе» («Report of the DSB task force on acquiring defense software commercially») выступила за широкое использование более современных моделей.

11. Mills, H. D., «Top-down programming in large systems», in *Debugging Techniques in Large Systems*, R. Rustin, ed. Englewood Cliffs, N. J.: Prentice-Hall, 1971.
12. Parnas, D. L., «On the design and development of program families», *IEEE Trans. on Software Engineering*, SE-2, 1 (March, 1976), pp. 1–9; Parnas, D. L., «Designing software for ease of extension and contraction», *IEEE Trans. on Software Engineering*, SE-5, 2 (March, 1979), pp. 128–138.
13. D. Harel, «Biting the silver bullet», *Computer* (Jan., 1992), pp. 8–20.
14. Основополагающие статьи по сокрытию информации: Parnas, D. L., «Information distribution aspects of design methodology», Carnegie-Mellon, Dept. of Computer Science, Technical Report (Feb., 1971); Parnas, D. L., «A technique for software module specification with examples», *Comm. ACM*, 5, 5 (May, 1972), pp. 330–336; Parnas, D. L. (1972). «On the criteria to be used in decomposing systems into modules», *Comm. ACM*, 5, 12 (Dec., 1972), pp. 1053–1058.
15. Идеи объектов были первоначально набросаны Хоаром (Hoare) и Дейкстрой (Dijkstra), но первым и наиболее влиятельным их развитием был язык Simula-67 Даля (Dahl) и Найгаарда (Nygaard).
16. Boehm, B. W., *Software Engineering Economics*, Englewood Cliffs, N.J.: Prentice-Hall, 1981, pp. 83–94; 470–472.
17. Abdel-Hamid, T., and S. Madnick, *Software Project Dynamics: An Integrated Approach*, ch. 19, «Model enhancement and Brooks's law». Englewood Cliffs, N.J.: Prentice Hall, 1991.
18. Stutzke, R. D., «A Mathematical Expression of Brooks's Law». In *Ninth International Forum on COCOMO and Cost Modeling*. Los Angeles: 1994.
19. DeMarco, T., and T. Lister, *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1987.
20. Pius XI, *Encyclical Quadragesimo Anno*, [Ihm, Claudia Carlen, ed., *The Papal Encyclicals 1903-1939*, Raleigh, N.C.: McGrath, p. 428.]
21. Schumacher, E. F., *Small Is Beautiful: Economics as if People Mattered*, Perennial Library Edition. New York: Harper and Row, 1973, p. 244.
22. Schumacher, op. cit., p. 34.
23. Заставляющий задуматься плакат на стене провозглашает: «Свобода прессы принадлежит тому, у кого она есть».

24. Bush, V., «That we may think», *Atlantic Monthly*, 176, 1 (April, 1945), pp. 101–108.
25. Кен Томпсон (Ken Thompson) из Bell Labs, создатель UNIX, рано осознал важность больших экранов для программирования. Он придумал способ получать 120 строк кода в два столбца на своей примитивной электронно-накопительной трубке Tektronix. Он держался за свой терминал, пока сменилось целое поколение быстрых трубок с маленьким экраном.

Фредерик Брукс

**Мифический человеко-месяц,
или Как создаются программные системы**

Перевел с английского *А. Логунов*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Научный редактор	<i>Д. Мамонов</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>Н. Викторова, Н. Сидорова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.07.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1200. Заказ 0000.