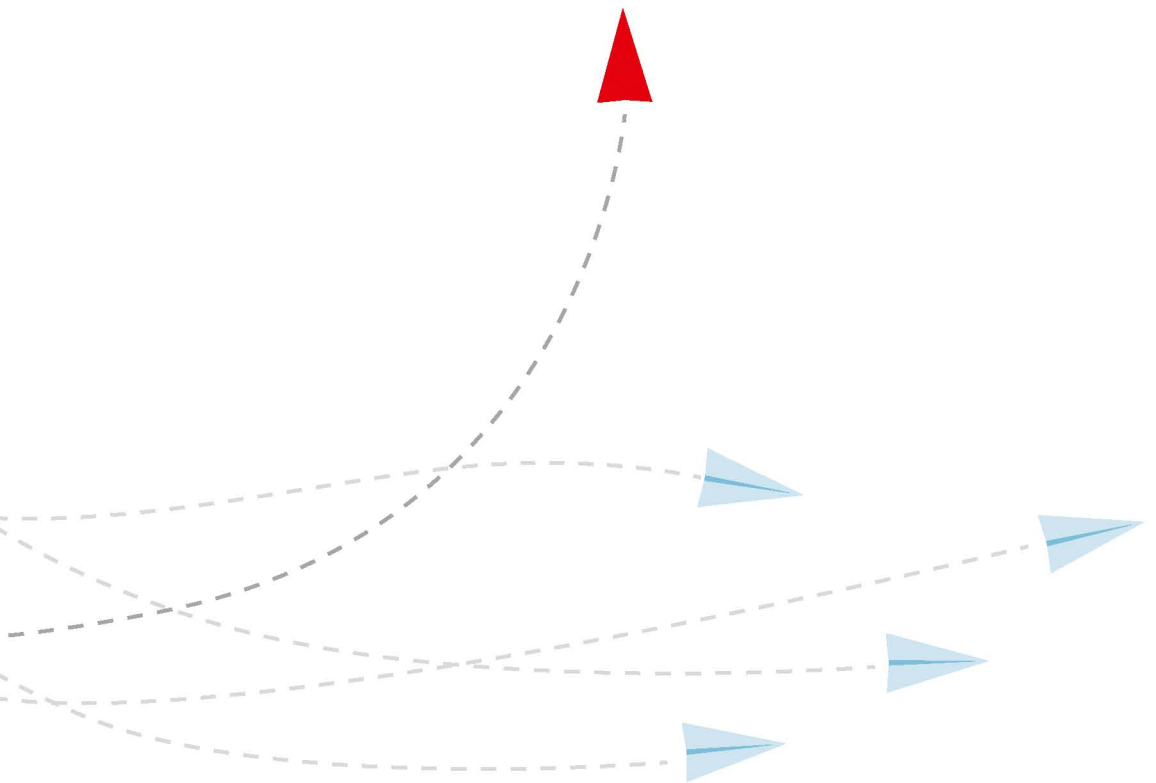
СОВЕРШЕННЫЙ СОФТ

[СОВЕРШЕННЫЙ СОФТ — СОВЕРШЕННАЯ КАРЬЕРА]



ДЖУВЕЛ ЛЁВЕ

Righting Software

A Method for System and Project Design

Juval Löwy

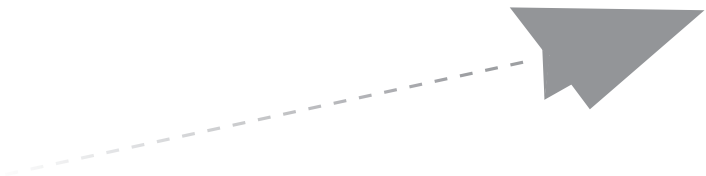
Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

ДЖУВЕЛ ЛЁВЕ

СОВЕРШЕННЫЙ СОФТ

[СОВЕРШЕННЫЙ СОФТ — СОВЕРШЕННАЯ КАРЬЕРА]



ПИТЕР®

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

2021

Джувел Лёве

Совершенный софт

Серия «Для профессионалов»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Макурина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*

ББК 32.973.2-018
УДК 004.4

Лёве Джувел

Л35 Совершенный софт. — СПб.: Питер, 2021. — 480 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1621-8

Совершенный софт — это проверенный, структурированный и высокотехнологичный подход к разработке программного обеспечения. Множество компаний уже используют идеи Лёве в сотнях систем, но раньше эти мысли нигде не публиковались.

Методология Лёве объединяет разработку систем и дизайн проектов, используя базовые принципы разработки ПО, корректные наборы инструментов и эффективные методы. Автор подробно описывает основы, на которых прокалываются многие архитекторы ПО, и показывает, как разложить систему на мелкие блоки или службы. Вы узнаете, как вывести эффективный дизайн проекта из дизайна системы, как рассчитать время, необходимое на запуск проекта, его стоимость и риски и даже как разработать несколько вариантов выполнения.

Метод и принципы совершенного софта можно применять независимо от размера проекта, компании, технологии, платформы или отрасли. Цель этой книги — решение важнейших задач современной разработки ПО, требующих исправления программных систем и проектов, ваш карьерный рост и, возможно, изменение всей IT-индустрии. Рекомендации и знания, которые вы получите, сэкономят десятилетия вашего опыта и спасут многие проекты. Эта книга принесет большую пользу разработчикам, архитекторам, руководителям проектов или менеджерам на любом этапе карьеры.

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

ISBN 978-0136524038 англ. Authorized translation from the English language edition, entitled RIGHTING SOFTWARE, 1st Edition by JUVAL LOWY, published by Pearson Education, Inc, publishing as Addison-Wesley Professional,

ISBN 978-5-4461-1621-8

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: 11.2020. 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 28.10.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1000. Заказ 0000.

Краткое содержание

ПРИЛОЖЕНИЯ

Оглавление

ЧАСТЬ I
ПРОЕКТИРОВАНИЕ СИСТЕМЫ

ЧАСТЬ II
ПЛАНИРОВАНИЕ ПРОЕКТА

ПРИЛОЖЕНИЯ

Отзывы о книге «Совершенный софт»

Я посещал как мастер-класс для архитекторов, так и мастер-класс по планированию проектов. До этого я почти потерял всякую надежду, что мне когда-нибудь удастся понять, почему работа моей команды никогда не приводит к успешному завершению, и лихорадочно пытался найти работоспособное решение, которое бы остановило нашу безумную гонку на выживание. Мастер-классы открыли передо мной мир, в котором разработка ПО поднимается на уровень всех остальных инженерных дисциплин и организуется профессионально, предсказуемо и надежно, что приводит к разработке высококачественных программных продуктов в рамках срока и бюджета. Полученные знания были бесценными! От создания основательной, проработанной архитектуры, которая может устоять перед вечно изменяющимися требованиям пользователей, до внутренних подробностей планирования и направления проекта к успешному завершению — все это было представлено с непревзойденным профессионализмом и компетентностью. А если учесть, что каждое слово истины, которым Джувел поделился с нами на мастер-классах, было получено, проверено и подтверждено реальной жизнью, этот усвоенный опыт превратился в мощный комплекс знаний, абсолютно необходимый для каждого, кто стремится стать архитектором в области программных систем.

Россен Тотев (Rossen Totev),
программный архитектор/руководитель проектов

Мастер-класс по планированию проектов стал событием, изменившим мою карьеру. Я вырос в среде, в которой ограничения по срокам и бюджетам нарушались патологически, так что возможность учиться у Джувела стала даром небес. Шаг за шагом он представлял компоненты и необходимые инструменты для правильного планирования проектов. Это позволяет держать под контролем затраты и сроки в динамической и даже хаотичной среде современной разработки программных продуктов. Джувел говорит, что вы вступаете в асимме-

тричную войну против превышения затрат и сроков, а после его мастер-классов у вас появляется ощущение, что вступаете в рукопашный бой с пистолетом. Никакого волшебства в этом нет — только применение базовых инженерных и производственных доктрин в области ПО, но, вернувшись в офис, вы будете чувствовать себя настоящим волшебником.

Мэтт Роболд (Matt Robold),
руководитель разработки ПО, West Covina Service Group

Фантастический опыт. Навсегда изменил мой образ мышления относительно того, как следует подходить к разработке программных продуктов. Я всегда знал, что какая-то часть моих ощущений относительно проектирования и программирования была правильной. Просто я не умел выразить ее словами, зато теперь эти слова у меня появились. Новые знания повлияли на мой подход не только к проектированию программных продуктов, но и к другим видам проектирования.

Ли Мессик (Lee Messick),
ведущий архитектор

Программный проект, над которым я работаю, годами страдал от немыслимо жестких сроков. Попытки разобраться в методологиях разработки и сформировать правильный процесс казались напрасной тратой сил, потому что помимо неразумных требований заказчиков приходилось бороться с нежеланием руководства идти на какие-либо компромиссы. Я сражался на двух фронтах, и поражение казалось неизбежным. Мастер-класс предоставил четкое видение ситуации, о котором я никогда не подозревал. Я получил именно те знания, которые искал. Освоил фундаментальные методы, которые изменили мое понимание того, как работают программные проекты. Теперь у меня есть инструменты для эффективного управления моими проектами в потоке бесконечно изменяющихся требований. В мире хаоса этот мастер-класс навел порядок. Я бесконечно благодарен IDesign. Моя жизнь никогда не будет прежней.

Аарон Фридман (Aaron Friedman),
архитектор

Жизнь изменилась. Я чувствую себя как хорошо настроенное пианино, на котором уже пару десятилетий копилась пыль.

Джордан Ян (Jordan Jan),
технический директор/архитектор

Курсы были просто фантастическими. Пожалуй, это была самая напряженная, но и самая плодотворная неделя моей профессиональной жизни.

Стоил Панков (Stoil Pankov), архитектор

Обучение у Джувела Лёве изменило мою жизнь. Я прошел путь от простого разработчика до настоящего архитектора, применяющего инженерные принципы из других дисциплин к проектированию не только программных продуктов, но и своей карьеры.

Кори Торгерсен (Kory Torgersen), архитектор

Мастер-класс для архитекторов — настоящий урок жизни из области проектирования и квалификации, который я прошел дважды. Знания преподносились на совершенно новом уровне, и я жалею, что не прошел это обучение десятки лет назад, в самом начале своей карьеры. «Перепрошивать мозг» и забывать то, что я знал ранее, неприятно, но я должен был пройти через это. Наконец, каждый последующий день я размышляю над тем, что сказал Джувел на мастер-классе, и пользуюсь новыми знаниями, помогая своей команде даже в мелочах — чтобы со временем все мы могли называть себя Профессиональными Инженерами. (P.S. При втором прохождении я написал более 100 страниц заметок!)

Джейсу Джеячандрат,
руководитель разработки ПО, Nielsen

Если вы расстроены, творческой энергии не осталось, если вы чувствуете отчаяние, видя множество неудач в нашей отрасли, этот мастер-класс вдохновит вас. Он поднимет вас на новый уровень профессиональной зрелости, а также придаст надежды и уверенности в том, что вы сможете правильно применить новые знания. Вы выйдете с мастер-класса по планированию проектов с новым образом мышления и достаточным количеством полезных инструментов, с которыми любая неудача программного проекта станет непростительной. Вы тренируетесь, решаете практические задачи, получаете представление о сути вещей и получаете опыт. Да, когда приходит время сообщать ключевым участникам данные о затратах, времени и риске проекта, МОЖНО быть точным. Не ждите, пока ваша компания отправит вас на этот мастер-класс. Если вы серьезно относитесь к своей карьере, поскорее запишитесь на этот или любой другой мастер-класс IDesign. Это лучшее вложение в свою карьеру, которое можно сделать. Спасибо всей команде IDesign за ее непрестанную работу по превращению отрасли разработки в серьезную инженерную дисциплину.

Люциан Мариан (Lucian Marian), архитектор, Mirabel

Как человек, который находится в начале своей карьеры, я могу честно заявить, что этот учебный курс изменил мою жизнь и мое отношение к работе. Я искренне считаю, что он стал поворотной точкой моей жизни.

Алекс Карпович (Alex Karpowich), архитектор

Хочу поблагодарить вас за неделю, изменившую мою (профессиональную) жизнь. Обычно я не могу высидеть на учебных курсах более 50% времени — они занудны и не учат меня ничему, чего бы я не знал или не мог узнать сам. На мастер-классе для архитекторов на протяжении 9 часов каждый день мне не было скучно: я узнал о своих обязанностях как архитектора (я-то думал, что архитектор — всего лишь проектировщик программных продуктов), об инженерном аспекте разработки, о важности реализации не только к заданному времени, но и в рамках бюджета и качества. Я научился не ждать, пока из меня «вырастет» архитектор, а активно управлять своей карьерой, научился оценивать и давать объективную оценку тому, что прежде считал интуитивными инсайтами. После этой недели многое встало на свои места. Не могу дождаться следующего мастер-класса.

Итай Золберг (Itai Zolberg), архитектор

Моему отцу
Томасу Чарльзу (Томми) Лёве

Предисловие

Вряд ли хоть кто-то из нас пришел в разработку программного обеспечения не по своей воле. Многие буквально влюбляются в программирование и решают, что хотят зарабатывать им на жизнь. Тем не менее между радужными представлениями о карьере и темной, угнетающей реальностью разработки лежит пропасть. Отрасль программирования в целом переживает глубокий кризис. Этот кризис оказался особенно острым из-за своей многомерности; нарушен буквально каждый аспект разработки:

- **Затраты.** Бюджет, выделенный для проекта, слабо связан с фактическими затратами на разработку системы. Многие организации даже не пытаются решить проблему с затратами — возможно, потому, что просто не знают, как к ней подойти, или потому, что это заставит их признать, что работа над системой окажется экономически неоправданной. Даже если затраты на первую версию новой системы будут оправданны, часто затраты на протяжении срока жизни системы намного выше положенных из-за некачественного проектирования и неспособности адаптироваться к изменениям. Со временем затраты на сопровождение становятся настолько неприемлемыми, что компании обычно решают начать с чистого листа — только для того, чтобы в ближайшем будущем получить не менее, а то и более дорогостоящую кучу хлама вместо новой системы. Ни в одной другой отрасли не встречается регулярный перезапуск систем просто из-за того, что поддержка старой системы экономически нецелесообразна. Авиакомпании эксплуатируют лайнеры десятилетиями, а жилой дом может быть построен сотню лет назад.
- **График.** Дедлайны часто оказываются взятыми из головы и являются ничем не обеспеченными конструкциями, так как не имеют никакого отношения ко времени, которое реально потребуется для разработки системы. Для большинства разработчиков дедлайны — всего лишь бесполезные ориентиры, которые проносятся мимо в процессе работы. Если команда разработки

выдерживает дедлайн, все удивляются, ибо всегда ждут, что сроки будут сорваны. И это тоже является прямым результатом плохого проектирования системы, что приводит к каскадному распространению в ней новых изменений и новой работы, с которой ранее завершенная работа становится недействительной. Более того, это является результатом чрезвычайно неэффективного процесса разработки, который не учитывает зависимости между активностями и не пытается отыскать самый быстрый и безопасный способ построения системы. Мало того что время реализации всей системы становится неприемлемо большим — и время реализации отдельных функций тоже может увеличиваться. Нарушение сроков в проекте — достаточно плохо; но еще хуже, когда нарушение скрывается от руководства и заказчиков, потому что никто не имеет ни малейшего понятия об истинном состоянии проекта.

- **Требования.** В конечном итоге часто оказывается, что разработчики решают не те задачи. Между заказчиками или их внутренними посредниками (например, отделом маркетинга) и группой разработки существует постоянное недопонимание. Также многие разработчики не признают свою неспособность сохранить суть требований. Даже идеально переданные требования с большой вероятностью со временем изменятся. Такие изменения делают решение недействительным и ставят под удар все, что пыталась построить команда.

- **Персонал.** Даже среднестатистические программные системы настолько сложны, что человеческий мозг просто не способен в них полностью разобраться. Внутренняя и внешняя сложность является прямым результатом неудачной архитектуры системы; в свою очередь, она ведет к появлению запутанных систем, создающих массу сложностей в сопровождении, расширении и повторном использовании.

- **Сопровождение.** Часто сопровождением программных систем занимаются совсем не те люди, которые их разрабатывали. Новые сотрудники не понимают, как работает система, и в результате попытки решения старых проблем постоянно приводят к созданию новых. Все это ведет к стремительному росту затрат на сопровождение и времени вывода продукта на рынок, а также к отмене проектов или их перезапуску с чистого листа.

- **Качество.** Возможно, ничто не нарушается в программных системах так серьезно, как качество. В программных продуктах есть баги, и можно сказать, что они стали неотъемлемой частью разработки. Говоришь «ПО» — имеешь в виду «баги». Разработчики не могут представить себе программную систему, избавленную от багов. Исправление дефектов (как и добавление новых функций или просто сопровождение) часто увеличивает счетчик ошибок. Плохое качество является прямым результатом малопонятной системной

архитектуры, которая создает проблемы с тестированием и сопровождением. Не менее важно то, что многие проекты не учитывают важнейшие активности по контролю качества и не выделяют достаточно времени для того, чтобы каждая активность была завершена идеально.

Несколько десятилетий назад в отрасли начали разрабатываться продукты для решения мировых проблем. В наши дни разработка сама по себе стала проблемой мирового уровня. Проблемы разработки часто проявляются в нетехнических аспектах: рабочих средах с высоким уровнем стресса, высокой текучести кадров, нервном истощении, отсутствии доверия, низкой самооценке и даже в соматических заболеваниях.

Ни одна из проблем разработки не нова[1]. Некоторые люди за всю свою карьеру в разработке ни разу не видели, чтобы продукт был сделан правильно. Они начинают верить, что это в принципе невозможно, и отвергают любые попытки решения проблем, потому что «уж так сложилось». Они даже могут оказывать сопротивление людям, пытающимся улучшить процесс разработки. Они уже решили для себя, что цель недостижима, поэтому каждый, кто пытается добиться лучших результатов, желает невозможного, а это оскорбляет их самолюбие.

Мой опыт является контрпримером, который доказывает, что успешная разработка программных систем возможна. Каждый проект, за который я отвечал, выпускался в пределах срока, бюджета и без дефектов. История продолжилась и после основания компании IDesign, в которой мы помогаем своим клиентам снова и снова успешно выполнять свои обязательства.

Эта последовательная, систематичная история успеха не была случайна. Я получал образование в области проектирования систем — как физических, так и программных, поэтому было нетрудно узнать сходство этих двух миров. Применение практических принципов — идей, которые считаются проявлением здравого смысла в других инженерных областях, — было оправданно и в области программных систем. Мне никогда не приходило в голову рассматривать разработку программного продукта как техническую задачу или разрабатывать систему без плана. Я не видел смысла идти на компромиссы со своими убеждениями или поступаться принципами, потому что правильный подход работал, а устрашающие последствия отхода от него были очевидны. Мне повезло с учителями; я оказался в нужном месте в нужное время, видел, что работает, а что не работает; мне представилась возможность поучаствовать в больших важных проектах и быть частью культуры высших достижений.

В последние годы я заметил, что проблемы отрасли усугубляются. Все больше программных проектов завершаются неудачей. Эти неудачи обходятся все до-

[1] Edsger W. Dijkstra. The Humble Programmer: ACM Turing Lecture. Communications of the ACM 15, no. 10 (October 1972): 859–866. (*Дейкстра Эдсгер*. Смиренный программист.)

роже по времени и деньгам, и даже завершенные проекты обычно отклоняются от своих исходных обязательств. Кризис усугубляется не только тем, что системы становятся все больше, не только жесткими сроками или более высокой частотой изменений. Подозреваю, настоящая причина заключается в том, что знания о том, как правильно проектировать и планировать программные системы, медленно уходят из команд разработчиков. Когда-то в большинстве команд присутствовал ветеран, который обучал молодых и передавал коллективные знания. В наши дни эти учителя перешли на другую работу или вышли на пенсию. В их отсутствие рядовые сотрудники располагают бесконечным количеством информации при нуле знаний.

Я бы очень хотел, чтобы кризис в области программирования мог быть решен каким-то одним способом: применением процесса, методологии разработки, инструмента или технологии. К сожалению, для решения многомерной задачи требуются многомерные решения. В этой книге предлагается комплексное решение: исправление процесса разработки.

По сути, все, что я предлагаю, — проектирование и разработка программных систем с применением инженерных принципов. К счастью, нет необходимости заново изобретать велосипед. Другие инженерные дисциплины были достаточно успешны, поэтому отрасль разработки программного обеспечения может позаимствовать ключевые универсальные принципы проектирования и адаптировать их для программных систем и проектов. Чтобы добиться успеха, необходимо принять инженерную точку зрения. Вы хотите, чтобы программная система была простой в сопровождении и расширении, экономичной, пригодной для повторного использования и реализуемой по времени и риску? Все это инженерные, а не технические аспекты. Они напрямую обусловлены проектированием системы и планированием проекта. Появился специальный термин «архитектор программного продукта» (или просто «архитектор»), которым обозначается участник команды, который отвечает за все аспекты проекта, связанные с проектированием. Соответственно я буду называть читателя «архитектором».

Идеи, представленные в книге, не затрагивают всего, что необходимо знать для достижения успеха. Тем не менее они безусловно станут хорошей отправной точкой, так как направлены на исправление корневой причины всех проблем, упоминавшихся ранее. Корневой причиной является некачественное проектирование самой программной системы или планирование проекта, используемого для построения этой системы. Вы увидите, что программный продукт вполне может быть реализован в пределах графика и бюджета и что планировать системы, удовлетворяющие всем мыслимым требованиям, реально. Такие системы также не создают проблем в сопровождении, расширении и повторном использовании. Надеюсь, что использование этих идей выведет на правильный путь не только систему, над которой вы работаете, но и вашу карьеру, и что оно снова разожжет вашу страсть к программированию.

Структура книги

В книге представлен структурированный инженерный подход к проектированию систем и планированию проектов. Методология состоит из двух частей, нашедших отражение в структуре книги: проектирование системы (создание архитектуры) и планирование проекта. Обе части взаимно дополняют друг друга и являются обязательными. Приложения дополняют основной материал.

Обычно в технической литературе каждая глава посвящена отдельной теме. Такой подход упрощает написание книги, но обычно не соответствует особенностям нашего процесса познания. Напротив, в этой книге процесс обучения напоминает спираль. В обеих частях книги в каждой главе мы возвращаемся к идеям предыдущих глав, погружаясь в материал или развивая идеи на основании дополнительной информации, охватывающей разные аспекты. Такая подача материала повторяет естественный процесс обучения. Каждая глава строится на материале предшествующих глав, поэтому читать главы стоит по порядку. В обе части книги включен подробный практический пример, демонстрирующий основные идеи, а также дополнительные аспекты. В то же время, чтобы итерации были более компактными, обычно я стараюсь избегать самоповторений, так что даже ключевые моменты обсуждаются только один раз.

Ниже приведено саммари глав и приложений.

Глава 1. Метод

В главе 1 представлена ключевая идея: чтобы добиться успеха, вы должны спроектировать систему и спланировать проект для ее построения. Обе составляющие крайне важны для успеха. Проект невозможно планировать без архитектуры, а строить систему, которую вы не сможете построить, бессмысленно.

Часть I. Проектирование системы

Глава 2. Декомпозиция

Глава 2 посвящена разложению системы на компоненты, образующие ее архитектуру. Большинство архитекторов выполняет декомпозицию систем худшим из всех возможных способов, поэтому глава начинается с объяснения того, чего делать не следует. Разобравшись с ошибками, вы увидите, как правильно выполнить декомпозицию системы, и освоите простые средства анализа и наблюдения, которые помогут вам в этом процессе.

Глава 3. Структура

Глава 3 развивает идеи главы 2 и дополняет их структурой. Вы узнаете, как сохранять требования, как строить иерархию уровней в архитектуре, освоите

таксономию компонентов архитектуры, их взаимоотношения, узнаете конкретные рекомендации по классификации, а также некоторые сопутствующие аспекты, такие как проектирование подсистем.

Глава 4. Композиция

В главе 4 показано, как объединить компоненты системы в композицию, которая удовлетворяет заданным требованиям. В этой короткой главе приводятся некоторые из ключевых принципов книги; она использует материал двух предыдущих глав для создания мощного ментального инструмента, который вы будете применять в любой системе.

Глава 5. Пример проектирования системы

Глава 5 содержит обширный практический пример, который демонстрирует основные идеи проектирования систем, рассматривающиеся до этого момента. Последняя итерация спирали проектирования системы представляет реальную систему, выравнивает проектировочное решение с требованиями бизнеса и показывает, как создать архитектуру и проверить ее на жизнеспособность.

Часть II. Планирование проекта

Глава 6. Мотивация

Так как большинство людей никогда не слышало о планировании проектов (не говоря уже о том, чтобы применять его), в этой главе описана эта концепция и предоставлены побудительные причины для участия в планировании проекта. Это нулевая итерация спирали планирования проектов.

Глава 7. Обзор планирования проекта

В главе 7 приведен общий обзор планирования проектов. Глава начинается с определения успеха при разработке, после чего представляются ключевые концепции обоснованных решений, комплектования проекта, сети проекта, критического пути, сроков и затрат. В главе рассматриваются многие идеи и методы, используемые в последующих главах, а завершается она важным обсуждением ролей и обязанностей.

Глава 8. Сеть и временные резервы

В главе 8 подробно рассматривается сеть проекта и ее использование как инструмента проектирования. Вы увидите, как смоделировать проект в качестве диаграммы сети, узнаете ключевую концепцию временного резерва, научитесь

использовать временные резервы при комплектовании и назначении сроков, а также поймете, как временные резервы связаны с риском.

Глава 9. Время и затраты

В главе 9 определяются возможные компромиссы между временем и затратами в любых проектах, а также описываются возможности ускорения проектов за счет чистой и правильной организации работы. Кроме этого, вы познакомитесь с ключевыми концепциями уплотнения, кривой «время-затраты» и составляющими затрат.

Глава 10. Риск

В главе 10 представлен отсутствующий элемент большинства проектов: риск, выраженный в числовой форме. Вы узнаете, как измерить риск, как связать его с концепциями времени и затрат из предыдущей главы и как вычислить риск на основании сети. Риск часто становится лучшим способом оценки вариантов, это первоклассный инструмент планирования.

Глава 11. Планирование проекта в действии

В главе 11 все концепции предыдущих глав реализуются на практике посредством систематического применения шагов, задействованных в планировании проекта. Хотя проект обладает всеми качествами примера, он приведен прежде всего для демонстрации процесса мышления, используемого при планировании проектов, а также подготовки представления проекта ответственным за принятие решений со стороны бизнеса.

Глава 12. Расширенные методы планирования

В соответствии со спиральной моделью обучения в этой главе приводится описание нетривиальных приемов и концепций. Эти методы применяются в проектах разных уровней сложности, от простых до самых сложных. Эти расширенные методы дополняют предыдущие главы и друг друга и часто применяются в сочетании друг с другом.

Глава 13. Пример планирования проекта

В главе 13 рассматривается пример планирования проекта, соответствующего примеру проектирования системы из главы 5. Рассматриваемый проект также является практическим примером, демонстрирующим процесс планирования проекта от начала до конца. Пример занимает в этой главе центральное место, а методам отводится вторичная роль.

Глава 14. Завершение

Последняя глава слегка отступает от технических аспектов планирования. В ней предлагается подборка рекомендаций, советов, точек зрения и идей процесса разработки. Глава начинается с ответа на важный вопрос: когда следует заниматься планированием проекта? А в конце главы рассматривается влияние плана проекта на качество.

Приложения

Приложение А. Отслеживание проекта

Приложение А показывает, как отслеживать прогресс проекта с учетом плана и как принимать меры по исправлению ситуации, когда это потребуется. Отслеживание проекта в большей степени направлено на управление проектом, нежели на планирование проектов, но оно критично для выполнения ваших обязательств после начала работы.

Приложение Б. Проектирование контрактов сервисов

Архитектура сама по себе достаточно широка и неопределенна, и вам придется спроектировать подробности каждого из ее компонентов. Самая важная из этих подробностей — контракты сервисов. В приложении Б обозначен правильный путь проектирования контрактов сервисов. Кроме того, обсуждение модульности, размера и затрат очень хорошо сочетается с большинством глав этой книги.

Приложение В. Стандарт проектирования

В приложении В приведен объединенный список ключевых директив, рекомендаций, а также того, что можно или чего нельзя делать в тех или иных ситуациях. Стандарт предельно лаконичен; в нем можно найти ответы на вопрос «что?», а не «почему?». Обоснования стандарта содержатся в основном тексте книги.

Кто вы?

Хотя книга написана для архитекторов в области разработки программных систем, она имеет намного более широкую аудиторию. Предполагается, что вы — читатель — являетесь архитектором или старшим специалистом в области разработки, менеджером проекта или совмещаете сразу несколько ролей.

Разработчики-энтузиасты, желающие повысить свою квалификацию, найдут в книге много полезного. Впрочем, независимо от вашей текущей должности книга откроет перед вами немало дверей на протяжении всей вашей карьеры. Даже если вы не являетесь опытным архитектором, открывая первую страницу книги, после ее прочтения и освоения методологии вы будете стоять наравне со специалистами высочайшего уровня.

Методы и идеи, представленные в книге, актуальны независимо от языка программирования (будь то C++, Java, C# или Python), платформы (Windows, Linux, мобильные, локальные и облачные) и размера проекта (от самых мелких до самых крупных проектов). Также они актуальны для всех отраслей (от здравоохранения до обороны), любых бизнес-моделей и размеров компаний (от начинающих фирм до крупных корпораций).

Самое важное предположение, которое я делаю относительно читателя, — что вам действительно небезразлично то, чем вы занимаетесь, а текущие неудачи и потери вас беспокоят. Вы хотите добиться большего, но не знаете, что делать, или вас сбивают с толку некачественные практики.

Что необходимо для работы с книгой

Единственное обязательное условие — непредвзятый ум. Прошлые неудачи и огорчения весьма желательны.

Дополнительные онлайн-ресурсы

На веб-странице книги представлены файлы примеров, дополнения и исправления ошибок. Страница доступна по адресу

http://www.rightingsoftware.org

Вы найдете файлы и вспомогательный материал для книги по ссылке Download Support Files.

За дополнительной информацией о книге обращайтесь по адресу

informit.com/title/9780136524038

Также с автором можно связаться по адресу

http://www.idesign.net

Благодарности

Прежде всего, хочу поблагодарить двух друзей, которые убедили меня написать эту книгу, причем каждый по-своему: Гад Меир (Gad Meir) и Яркко Кемппайнен (Jarkko Kemppainen).

Спасибо редактору и по совместительству источнику объективной критики Дэйву Киллиану (Dave Killian): еще немного правки, и мне пришлось бы указывать тебя как соавтора. Спасибо Бет Сайрон (Beth Siron) за рецензирование чернового варианта рукописи. Следующие люди выделили свое время на рецензирование рукописи: Чед Майкл (Chad Michel), Дуг Дархэм (Doug Durham), Джордж Стивенс (George Stevens), Джош Лойд (Josh Loyd), Риккардо Беннет-Ловси (Riccardo Bennett-Lovsey) и Стив Лэнд (Steve Land).

Наконец, я благодарен своей жене Дане, которая вдохновляет меня писать книги и помогает выделить время, освобождая меня от семейных обязанностей. И спасибо моим родителям, привившим мне любовь к техническим дисциплинам.

Об авторе

Джувел Лёве, основатель IDesign, — старший архитектор программных систем, специализирующийся на проектировании систем и планировании проектов. Помогал бесчисленным компаниям по всему миру создавать качественные продукты в рамках сроков и бюджета. Признанный компанией Microsoft как один из ведущих экспертов и отраслевых лидеров, он принимал участие во внутреннем стратегическом анализе архитектуры C#, WCF и других сопутствующих технологий и был прозван «легендой от программирования». Опубликовал несколько книг и множество статей практически по всем аспектам современной разработки. Лёве часто выступает на крупных международных конференциях разработчиков и проводит мастер-классы по всему миру. Он обучает тысячи профессионалов навыкам, необходимым для современных архитекторов программных систем, и лидерству, нужному в проектировании, процессах и технологиях.

От издательства

Цветные версии рисунков вы можете посмотреть, отсканировав QR-код.

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.

Для начинающего архитектора существует множество вариантов решения практически любой задачи.

Для опытного архитектора хороших вариантов совсем немного.

1 Метод

«Дзен архитектора»[1], по сути, гласит, что для начинающего архитектора существует множество вариантов решения практически любой задачи. Однако для опытного архитектора хороших вариантов совсем немного — как правило, всего один.

Начинающие архитекторы часто приходят в замешательство от изобилия паттернов, идей, методологий и возможностей проектирования их программных систем. В отрасли разработки полным-полно идей и людей, которые желают учиться и самосовершенствоваться, — включая вас, читателя этой книги. Но поскольку существует лишь несколько правильных подходов к любой задаче по проектированию, с таким же успехом вы можете сосредоточиться на них и не обращать внимания на посторонний шум. Мастера программной архитектуры знают, как это делается; словно под воздействием какого-то сверхъестественного вдохновения, они немедленно проникают в суть дела и выдают правильно спроектированное решение.

«Дзен архитектора» применим не только к архитектуре систем, но и к проекту, в котором она строится. Да, существует бесчисленное множество способов структурирования проекта и распределения работы между членами команды, но все ли они в равной степени безопасны, быстры, экономичны, полезны, эффективны и действенны? Опытный архитектор также планирует проект для построения системы и даже помогает руководству принять исходное решение о том, может ли оно вообще себе позволить этот проект.

Истинное мастерство в любой области — это путь. Никто не рождается экспертом (за редчайшим исключениями). Моя карьера наглядно свидетельствует об этом. Я начал работать младшим архитектором почти 30 лет назад, когда сам термин «архитектор» еще не так часто встречался в фирмах, занимающихся разработкой. Сначала я перешел на роль архитектора проекта, потом на роль архитектора подразделения, а к концу 1990-х стал ведущим архитектором про-

[1] https://en.wikipedia.org/wiki/Zen_Mind,_Beginner's_Mind. (На русском: *Сюнрю Судзуки*. Сознание дзен, сознание начинающего. — *Примеч. ред.*)

граммных систем в компании из Кремниевой долины из списка Fortune 100. В 2000 году я основал IDesign — компанию, чья деятельность посвящена исключительно проектированию программных систем. С тех пор в IDesign были спроектированы сотни систем и проектов. Хотя в каждом случае существовала конкретная архитектура и план проекта, независимо от заказчика, проекта, системы, технологии или состава разработчиков мои рекомендации по проектированию, по сути, оставались одними и теми же.

И тогда я задал себе простой вопрос: действительно ли необходимо быть опытным архитектором с десятилетиями опыта практического проектирования программных систем и десятками проектов за плечами, чтобы выбрать правильный подход? Или возможно структурировать проект так, чтобы любой человек с четким пониманием используемой методологии смог предложить достойный вариант дизайна системы и проекта?

На второй вопрос я отвечаю решительным «да». Результат, который я называю «Методом», и является темой этой книги. В результате применения Метода во множестве разных проектов, после обучения и курирования нескольких тысяч архитекторов по всему миру я убедился в том, что при правильном применении он работает. При этом я не умаляю ценности правильного отношения, технической квалификации и аналитических способностей — все это необходимые ингредиенты успеха независимо от используемой методологии. К сожалению, этих ингредиентов недостаточно; я часто вижу, как проекты проваливаются, хотя их разработчики обладают всеми замечательными качествами и атрибутами. Однако в сочетании с Методом эти ингредиенты дают шанс на успех. Если ваш дизайн будет базироваться на надежных инженерных принципах, вы научитесь держаться подальше от порочных практик и ложных интуитивных представлений, которые так часто принимают за житейскую мудрость.

Что такое «Метод»?

Метод — простая и эффективная система анализа и проектирования. Суть Метода можно выразить формулой:

Метод = проектирование системы + планирование проекта.

В том, что касается проектирования системы, Метод закладывает основу для разбиения большой системы на меньшие модульные компоненты. Метод предоставляет рекомендации по выбору структуры, роли и семантики компонентов и по организации взаимодействий между ними. Результат представляет собой архитектуру системы.

При планировании проекта Метод помогает сформулировать различные варианты построения системы. Каждый вариант представляет собой сочетание графика, затрат и рисков. Также каждый вариант определяет набор инструкций

по сборке системы и готовит проект к выполнению и дальнейшему отслеживанию.

Планирование проекта, рассматриваемое в части II книги, является намного более важным фактором успеха, чем проектирование системы. Даже посредственно спроектированная система может оказаться успешной, если проект располагает достаточным временем и ресурсами и если риск находится на приемлемом уровне. При этом систему, спроектированную на первоклассном уровне, ждет провал, если у проекта не хватит времени или ресурсов для ее построения или если проект окажется слишком рискованным. Планирование проекта также намного сложнее проектирования системы и, как следствие, требует большего арсенала инструментов, идей и приемов.

Поскольку Метод сочетает проектирование системы с планированием проекта, фактически он может рассматриваться как процесс проектирования. За прошедшие годы отрасль разработки уделяла огромное внимание процессу разработки и намного меньшее — процессу проектирования. Эта книга призвана заполнить пробел.

Проверка результата проектирования

Проверка результата проектирования также чрезвычайно важна — организация не может рисковать, поручая команде начать разработку непроработанной архитектуры или системы, на построение которой может не хватить средств. Метод оказывает содействие и поддержку в решении этой критически важной задачи, позволяя архитектору утверждать с разумной степенью уверенности, что предложенное решение адекватно, то есть что оно удовлетворяет двум ключевым целям. Во-первых, решение должно удовлетворять потребности заказчика. Во-вторых, оно должно соответствовать возможностям и ограничениям организации или команды.

Когда команда займется программированием, изменение архитектуры часто становится неприемлемым из-за возможных последствий для затрат и графика. На практике это означает, что без проверки результата проектирования системы возникает риск закрепления архитектуры в лучшем случае несовершенной, а в худшем — безобразной. Организации придется каким-то образом жить с полученной системой в течение нескольких следующих лет и нескольких версий до следующей крупной переработки. Плохо спроектированная программная система может серьезно повредить бизнесу, лишив его возможности реагировать на коммерческие возможности и даже причиняя серьезный финансовый ущерб из-за растущих затрат на сопровождение.

Ранняя проверка результата проектирования — первоочередная необходимость. Например, если через три года после начала работы выяснится, что конкретная идея или вся архитектура неверна, это будет теоретически интересно, но практическая ценность такого открытия равна нулю. В идеальном случае

через неделю после начала проекта вы должны знать, выдержит архитектура или нет. Любая задержка только повысит риск ведения разработки с сомнительной архитектурой. В следующих главах подробно описано, как проверить результат проектирования системы.

Обратите внимание: я говорю именно о проектирования системы, о ее архитектуре, а не о детально разработанном плане. Детально разработанный план определяет для каждого компонента архитектуры ключевые артефакты реализации (интерфейсы, иерархии классов и контракты данных). Такие планы дольше строятся, создаются в ходе исполнения проекта и могут изменяться по мере построения или эволюции системы.

Аналогичным образом необходимо проверять результат планирования проекта. Срыв графика или бюджета (или и того и другого) на середине работы попросту недопустим. Если вы не сможете выполнить свои обязательства, это плохо отразится на вашей карьере. Следует заблаговременно проверить план своего проекта и убедиться в том, что имеющаяся команда способна успешно завершить работу над проектом.

Кроме формирования архитектуры и планов проекта, одной из целей Метода является устранение рисков для проекта, обусловленных проектированием. Проект не должен провалиться только из-за того, что архитектура оказалась слишком сложной для построения и сопровождения. Метод выявляет архитектуру действенно и эффективно и делает это за короткий период времени. То же можно сказать о планировании проекта: проект не должен завершиться неудачей, потому что для него изначально было выделено недостаточно времени или ресурсов. Эта книга показывает, как точно вычислить продолжительность проекта и затраты на него и как принимать обоснованные решения.

Аврал

При использовании Метода можно спроектировать систему за считаные дни, обычно от 3 до 5 дней; примерно столько же времени уйдет на планирование проекта. С учетом высоких целей (а именно формирования архитектуры системы и проектного плана новой системы) это может показаться недостаточным. У типичных бизнес-систем возможность перепроектирования обычно представляется только через несколько лет, так почему бы не потратить 10 дней на архитектуру? На фоне жизненного цикла системы, измеряемого в годах, пять лишних дней даже не назовешь ошибкой округления. Тем не менее дополнительное время проектирования часто не только не улучшает результат, но даже оказывает отрицательное воздействие.

Во многих рабочих средах планирование времени организовано ужасающе неэффективно, прежде всего из-за человеческой природы. Аврал, или кранч, заставляет вас (и других причастных) сконцентрироваться, выбрать приори-

теты и сформировать результат. Применять Метод следует быстро и решительно.

В общем случае проектирование занимает немного времени (по сравнению с реализацией). Строительные архитекторы берут почасовую оплату и часто работают над проектированием дома не более одной-двух недель. На строительство дома по проекту архитектора может уйти два-три года мучительной работы с подрядчиками, но архитектору на разработку архитектуры много времени не потребовалось.

Аврал также помогает избежать «украшательства» проекта. Закон Паркинсона[1] гласит, что работа заполняет все отпущенное на нее время. Если на проектирование, которое может быть выполнено за 5 дней, будет отведено 10 дней, то, скорее всего, архитектор проработает над ним все 10 дней. Лишнее время будет потрачено на проработку второстепенных аспектов, которые не добавляют ничего, кроме сложности; это обернется непропорциональным повышением затрат на реализацию и сопровождение в предстоящие годы. Ограничение времени проектирования заставляет вас выдать достаточно хорошее решение.

Преодоление аналитического паралича

Аналитический паралич — ситуация, в которой человек (или команда), способный, умный и даже работящий (как и большинство архитекторов программных систем), застревает в бесконечном цикле анализа, проектирования, инсайтов и возврата к продолжению анализа. Работа человека (или команды) фактически парализуется, и ни о каком результативном труде не может быть и речи.

Дерево решений

Состояние паралича возникает прежде всего из-за незнания дерева решений как для проектирования системы, так и для планирования проекта. Дерево решений — общая концепция, применимая ко всем задачам проектирования, а не только в программировании. Проектирование любой сложной сущности состоит из множества малых проектировочных решений, выстроенных иерархически в древовидную структуру. Каждое разветвление в дереве представляет отдельный путь, приводящий к новым, более детализированным решениям из области проектирования. Листья дерева представляют разные проектировочные решения для действующих требований. Каждый лист — отдельное, целостное и действительное решение, чем-то отличающееся от всех остальных листьев.

---

[1] Cyril N. Parkinson, «Parkinson's Law», *The Economist* (November 19, 1955).

Если человек или команда, ответственные за разработку проектировочного решения, плохо представляют правильное дерево решения, то начинают не от корня дерева, а с другого места. В какой-то момент последующее решение неизбежно нивелирует предыдущее; все решения, принятые между этими двумя точками, станут недействительными. Такое проектирование напоминает пузырьковую сортировку дерева решений. Так как количество операций при пузырьковой сортировке приблизительно равно квадратному корню количества элементов, потери будут значительными. Простая программная система, требующая приблизительно 20 решений из области проектирования системы и планирования проекта, потенциально может потребовать около 400 итераций, если вы не следуете дереву решений. Проведение такого количества встреч (даже распределенных по времени) приведет к параличу. Вряд ли у вас найдется время хотя бы для 40 итераций. Когда время, выделенное на проектирование и планирование, подойдет к концу, разработка начнется в незрелом состоянии системы и проекта. При этом находки, из-за которых решения из области проектирования станут недействительными, будут отложены до еще более худшего момента в будущем, когда с неправильными решениями уже будут связаны время, усилия и артефакты. Потери от ошибочных решений будут максимизированы.

Дерево проектировочных решений программной системы

Как выясняется, у многих программных бизнес-систем есть много общего, и по крайней мере контуры дерева решений у таких систем не только похожи, но и даже унифицированы. Естественно, листья у них различаются.

Метод предоставляет дерево решений для типичной бизнес-системы — как для проектирования системы, так и для планирования проекта. Только после того, как вы спроектируете систему, появляется смысл планирования проекта для построения этой системы. У каждой из этих областей — проектирования системы и планирования проекта — имеется собственное поддерево проектировочных решений. Метод будет руководить вашими перемещениями по дереву, начиная от корня, что позволит избежать переделок и переоценки предыдущих решений.

Один из самых полезных приемов усечения дерева решений — применение ограничений. Как подметил Фредерик Брукс[1], вопреки здравому смыслу или интуиции, худшая задача по проектированию — это чистый холст. Без ограничений процесс проектирования должен быть простым, верно? Нет. Чистый холст приведет в ужас любого архитектора. Существует бесконечное множе-

[1] Frederick P. Brooks Jr., *The Design of Design: Essays from a Computer Scientist* (Upper Saddle River, NJ: Addison-Wesley, 2010) (на русском: *Фредерик П. Брукс.* Проектирование процесса проектирования: записки компьютерного эксперта. — М.: Вильямс, 2012. — 464 с. — *Примеч. ред.*).

ство способов решить задачу неправильно или нарушить скрытые ограничения. Чем больше ограничений, тем проще задача проектировщика. Чем меньше свободы действий, тем более очевиден результат проектирования. В полностью ограниченной системе проектировать нечего: все устроено совершенно определенным образом. Так как при проектировании всегда существуют ограничения (явные или неявные), при отработке дерева проектировочных решений Метод устанавливает последовательно нарастающие ограничения для системы и проекта вплоть до точки, в которой начинается быстрое схождение и окончательное формирование результата.

Обмен информацией

Важным преимуществом Метода является обсуждение идей, относящихся к проектированию. После того как участники будут знакомы с устройством архитектуры и семантикой проектирования, Метод предоставляет возможность обмениваться идеями и точно передавать информацию о том, чего требует. Вы можете представить команде свой процесс мышления, лежащий в основе проектирования. Делитесь различными решениями и наблюдениями, которыми вы руководствовались при формировании архитектуры, четко и однозначно документируйте свои рабочие версии и полученные в итоге проектировочные решения.

Такой уровень четкости и прозрачности намерений проектировщика абсолютно необходим для жизнеспособной архитектуры. Хорошим решением может считаться то, которое было хорошо продумано, пережило стадию разработки и в итоге было воплощено в виде работающих компонентов на машинах заказчика. Вы должны уметь передавать замысел проектирования разработчикам и следить за тем, чтобы они серьезно относились к намерениям и концепциям, лежащим в его основе. Концепция проектирования должна подкрепляться анализом, исследованиями и курированием. Метод прекрасно проявляет себя с коммуникациями такого типа вследствие сочетания четко определенной семантики и структуры.

Не сомневайтесь: если разработчики, которым поручено построение системы, не понимают и не ценят ее проектное решение, они ее угробят. Сколько бы времени ни было потрачено на критический анализ кода или проекта, это не изменит ситуации. Целью критического анализа должно быть выявление непреднамеренных отклонений от архитектуры на как можно более ранней стадии.

Все сказанное остается истинным и тогда, когда наступает время передать план проекта руководителям проекта, руководству или другим ключевым участникам. Четкие, однозначные, объективно сопоставимые варианты — ключ к обоснованным решениям. Когда люди принимают неверные решения, часто это происходит из-за того, что они не понимают проект и невер-

но представляют себе его поведение. Строя правильные модели для проекта в отношении времени, затрат и рисков, архитектор делает возможным принятие правильных решений. Метод предоставляет правильную терминологию и метрики для простого, конкретного общения с ответственными за принятие решений. После того как руководители осознают возможности проектного решения, они станут его главными сторонниками и будут настаивать на том, чтобы работа велась именно так, а не иначе. Никакие вдохновенные речи не сравнятся по эффективности с простым набором диаграмм и чисел. Более того, план проекта важен не только на начальной стадии. В процессе работы вы можете пользоваться средствами планирования проектов для того, чтобы сообщать руководству о действенности и целесообразности изменений. Отслеживание проекта и управление изменениями рассматриваются в приложении А.

Помимо передачи информации о проектировочном решении разработчикам и руководству, Метод позволяет архитектору точно и легко сообщать информацию другим архитекторам. Информация, которая может быть получена в ходе критического анализа и рецензирования такого рода, бесценна.

Чем Метод не является

В 1987 году Брукс написал: «Серебряной пули не существует»[1]. И конечно, Метод ею не является. Применение Метода не гарантирует успеха и может только усложнить ситуацию, если Метод будет использоваться в отрыве от других аспектов проекта или просто ради самого процесса.

Метод не исключает творческого подхода со стороны архитектора и усилий по построению подходящей архитектуры. Архитектор все равно обязан проработать требуемое поведение системы. Архитектор все равно несет ответственность за неправильную реализацию архитектуры, или за то, что информация о ней не была донесена до разработчиков, или за то, что пока он направлял процесс разработки до сдачи продукта, ему не удалось избежать нарушений архитектуры — пусть и в условиях нарастающего давления. Более того, как показано в части II книги, архитектор должен выдать работоспособный план проекта, созданный на основе архитектуры. Архитектор должен произвести калибровку проекта с учетом доступных ресурсов, задействованных рисков и сроков. Выполнять планирование проекта просто ради ритуала бессмысленно. Архитектор должен устранить все смещения и выдать правильный набор исходных предпосылок планирования и оценок результатов.

---

[1] Frederick P. Brooks Jr., "*No Silver Bullet: Essence and Accidents of Software Engineering*," Computer 20, no. 4 (April 1987). (Данную статью можно найти в издании: *Брукс Фредерик.* Мифический человеко-месяц, или Как создаются программные системы. — СПб.: Питер, 2021. — 368 с.: ил. — *Примеч. ред.*).

Метод может предоставить хорошую отправную точку для проектирования системы и планирования проекта, а также список факторов, которых следует избегать. Однако Метод работает только в том случае, если вы добросовестно применяете его, не жалея времени и когнитивных усилий для сбора необходимой информации. Вы должны искренне переживать за процесс проектирования и результаты, которые будет получены в результате его применения.

ЧАСТЬ I

Проектирование системы

2 Декомпозиция

Программная архитектура представляет собой высокоуровневый результат проектирования и структуру программной системы. Хотя проектирование системы занимает немного времени и обходится недорого по сравнению с ее построением, исключительно важно правильно реализовать архитектуру. Если после того, как система будет построена, окажется, что архитектура несовершенна, ошибочна или просто не подходит для ваших целей, сопровождение и расширение системы будет обходиться чрезвычайно дорого.

Сущность архитектуры любой системы заключается в разбиении концепции системы как целого (будь то дом, компьютер или программная система) на составляющие. Хороший архитектор также предписывает, как эти компоненты должны взаимодействовать друг с другом во время выполнения. Акт выявления составляющих компонентов системы называется *декомпозицией системы*.

Правильная декомпозиция исключительно важна для проектирования. Ошибки в ходе декомпозиции приводят к неправильной архитектуре, которая обернется колоссальными проблемами в будущем, а в некоторых ситуациях систему приходится переписывать с нуля.

В прошлом такими структурными элементами были объекты C++, а позднее — компоненты COM, Java или .NET. В современных системах и в этой книге самой детализированной единицей архитектуры является сервис (в сервисно-ориентированном понимании). Однако технология, применяемая для реализации компонентов, и их подробности (интерфейсы, операции и иерархии классов) относятся к подробностям проектирования, а не к декомпозиции системы. Такие подробности могут изменяться, и эти изменения никак не повлияют на декомпозицию и архитектуру.

К сожалению, многие программные системы (а возможно, абсолютное большинство) проектируются неправильно; можно даже сказать, что они спроектированы худшим из всех возможных способом. Дефекты проектирования являются прямым результатом неправильной декомпозиции систем.

По этой причине данная глава начинается с объяснения того, почему некоторые распространенные способы декомпозиции принципиально неверны. Далее приводятся обоснования процесса декомпозиции, заложенного в основу Метода. Также будут представлены некоторые мощные и полезные приемы, которые могут применяться при проектировании системы.

О вреде функциональной декомпозиции

Функциональная декомпозиция разбивает систему на структурные элементы, определяемые функциональностью системы. Например, если система должна выполнять некий набор операций (например, выставление счетов, оплату и поставку товара), вы определяете сервисы `Invoicing`, `Billing` и `Shipping`.

Недостатки функциональной декомпозиции

Функциональная декомпозиция обладает множеством серьезных недостатков. Как минимум функциональная декомпозиция связывает сервисы с требованиями, потому что сервисы отражают требования. Любые изменения в требуемой функциональности приводят к изменению функциональных сервисов. Такие изменения неизбежно возникают с течением времени и приводят к болезненным будущим изменениям в системе, требуя выполнения запоздалой повторной декомпозиции, отражающей новые требования. Помимо дорогостоящих изменений в системе, функциональная декомпозиция препятствует повторному использованию и ведет к появлению слишком сложных систем и клиентов.

Препятствия к повторному использованию

Возьмем простую систему с функциональной декомпозицией, использующую три сервиса, A, B и C, которые вызываются в следующем порядке: сначала A, затем B, и затем C. Так как функциональная декомпозиция также совпадает с декомпозицией по времени (сначала A, потом B), она фактически препятствует отдельному повторному использованию сервисов. Предположим, другой системе также необходим сервис B (допустим, `Billing`). В основу сервиса B встроено представление о том, что он вызывается после A, но до C (например, сначала выставляется счет, затем производится оплата, и только после этого происходит отгрузка). Любая попытка взять сервис B из первой системы и перенести его во вторую систему завершится неудачей, потому что во второй системе никто не выполняет A до, а C после B. При перенесении B к нему привязываются сервисы A и C. B не является независимым сервисом, пригодным для повторного использования, — A, B и C образуют конгломерат из тесно связанных сервисов.

Слишком много или слишком мало

Один из способов выполнения функциональной декомпозиции — создание сервисов для всех разновидностей функциональности. Этот способ декомпозиции приводит к взрывному росту количества сервисов, так как система сколько-нибудь приличного размера может содержать сотни видов функциональности. Вы не только получите слишком много сервисов, но и в этих сервисах часто будет дублироваться большой объем общей функциональности, адаптированной для конкретного случая. Размножение сервисов приводит к непропорционально высоким затратам на интеграцию и тестирование, а также к повышению общей сложности.

Другой способ функциональной декомпозиции основан на объединении всех возможных способов выполнения операций в «мегасервисы». Это приводит к разбуханию сервисов, в результате чего они становятся слишком сложными, а их сопровождение — практически невозможным. Такие монолиты превращаются в уродливые свалки для всех взаимосвязанных разновидностей исходной функциональности вместе с неочевидными отношениями внутри сервисов и между ними.

Таким образом, функциональная декомпозиция часто ведет к тому, что сервисы становятся слишком большими (тогда их слишком мало) или слишком мелкими (тогда их слишком много). Обе напасти нередко встречаются одновременно в одной системе.

> **ПРИМЕЧАНИЕ** В приложении Б, посвященном проектированию контрактов сервисов, более подробно обсуждаются неприятные последствия от слишком малого или слишком большого количества сервисов, а также от их влияния на проект.

Разбухание клиентов и связи

Функциональная декомпозиция часто приводит к уплощению системной иерархии. Поскольку каждый сервис или структурный элемент посвящен конкретной функциональности, кто-то должен объединить эти разрозненные функциональности в требуемое поведение. Этим «кем-то» нередко становится клиент. Когда клиент координирует работу сервисов, вы получаете плоскую систему из двух уровней: клиенты и сервисы; все промежуточные прослойки при этом исчезают. Допустим, ваша система должна выполнять три операции (или области функциональности): A, B и C, именно в таком порядке. Как показано на рис. 2.1, клиент должен «сшить» эти сервисы в единое целое.

Набивая клиент логикой координации, вы загрязняете клиентский код бизнес-логикой системы. Роль клиента уже не ограничивается вызовом операций в системе или представлением информации для пользователей. Клиент теперь

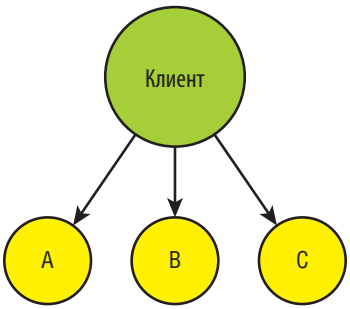


Рис. 2.1. Координация функциональности из раздувшегося клиента

должен знать все внутренние сервисы до мелочей: как вызывать их, как обрабатывать ошибки, как компенсировать сбой B после успеха A и т. д. Вызов сервисов почти всегда осуществляется синхронно, потому что клиент идет по ожидаемой последовательности «A, потом B, потом C», и другим способом было бы трудно обеспечить последовательность вызовов и обеспечить быструю реакцию на внешние события. Более того, клиент теперь жестко связывается с необходимой функциональностью. Любые изменения в операциях (скажем, вызов B' вместо B) должны быть отражены в клиенте. В идеале клиент и сервисы должны быть способны эволюционировать независимо друг от друга. Десятилетия назад программисты обнаружили, что включение бизнес-логики на стороне клиента почти всегда нежелательно. Тем не менее при таком проектировании, как на рис. 2.1, вы вынуждены загрязнять клиента бизнес-логикой соблюдения последовательности, упорядочения, компенсации ошибок и продолжительности вызовов. В конечном счете клиент перестает быть клиентом — он становится системой.

А что, если разнотипные клиенты (например, толстые (rich) клиенты, вебстраницы, мобильные устройства) пытаются вызвать одну и ту же последовательность функциональных сервисов? Вам неизбежно придется дублировать логику между клиентами, что сделает их сопровождение слишком неэффективным и дорогостоящим. С изменением функциональности вам придется синхронизировать изменения между разными клиентами, поскольку они отразятся на всех клиентах. Часто в таких ситуациях разработчики стараются избежать любых изменений в функциональности сервисов из-за каскадного воздействия на клиентов. Если вы создали множество клиентов, каждый из которых имеет собственную версию последовательности вызовов, адаптированную для его потребностей, с изменением или заменой сервисов возникает еще больше проблем, что препятствует повторному использованию поведения между клиентами. По сути, приходится заниматься сопровождением нескольких сложных систем, пытаясь синхронизировать их. В конечном итоге это приводит как к торможению нововведений, так и к затягиванию выпуска

новой версии при принудительном введении изменений в разработку и производство.

В качестве иллюстрации проблем функциональной декомпозиции, рассмотренных выше, возьмем рис. 2.2. На нем представлена схема анализа цикломатической сложности одной системы, которой я занимался. В качестве методологии проектирования была избрана функциональная декомпозиция.

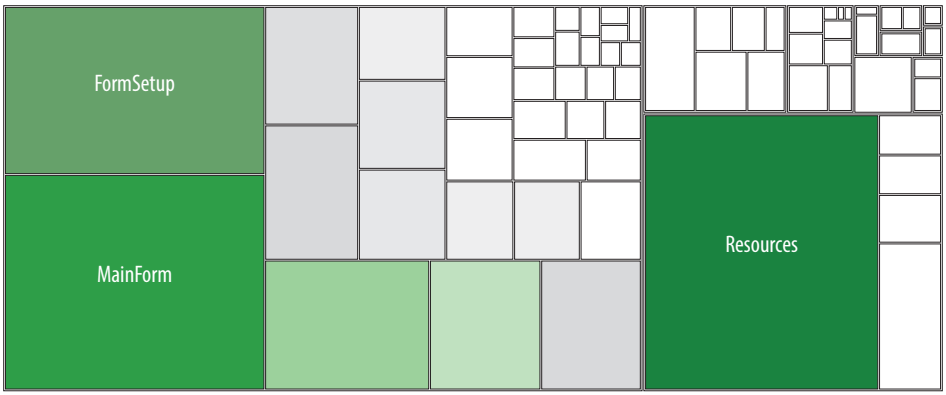


Рис. 2.2. Анализ сложности результатов функционального проектирования

Цикломатическая сложность оценивает количество независимых путей в коде класса или сервиса. Чем сложнее устроены и теснее связаны между собой внутренние механизмы, тем выше показатель цикломатической сложности. Программа, использованная для построения рис. 2.2, измерила и построила оценки для различных классов в системе. Чем сложнее класс, тем он больше и темнее на этой схеме. С первого же взгляда мы видим три очень больших и очень сложных класса. Насколько легко будет сопровождать `MainForm`? Что это — форма, элемент пользовательского интерфейса, канал для взаимодействия пользователя с системой или целая система? Обратите внимание на сложность, необходимую для настройки `MainForm`, — она отражена в размере и цвете `FormSetup`. Не отстает от них и класс `Resources` — изменение ресурсов, используемых в `MainForm`, оказывается очень сложным делом.

В идеале класс `Resources` устроен тривиально: он должен представлять собой простые списки изображений и строк. Остальная часть системы строится из десятков меньших простых классов, каждый из которых посвящен конкретной функциональности. Меньшие классы буквально оказались в тени трех огромных классов. Тем не менее, хотя каждый из этих меньших классов может быть тривиальным, огромное количество меньших классов само по себе создает проблему сложности из-за тонкостей интеграции между всеми классами. В результате мы имеем и слишком мелкие компоненты, и слишком крупные компоненты, и разбухший клиент.

Множество точек входа

Еще один недостаток декомпозиции на рис. 2.1 — необходимость множества точек входа в систему. Клиент (или клиенты) должен входить в систему в трех местах: для A, для B и для C. Это означает, что вам придется в нескольких местах беспокоиться об аутентификации, авторизации, масштабируемости, управлении экземплярами, распространении транзакций, идентификации, размещении и т. д. Если вам понадобится изменить реализацию любого из этих аспектов, придется изменять ее в разных местах по всем сервисам и клиентам. Со временем из-за этих многочисленных изменений добавление новых клиентов будет обходиться очень дорого.

Разбухание сервисов и связи

В качестве альтернативы последовательности функциональных сервисов на рис. 2.1 можно рассмотреть то, что на первый взгляд является меньшим злом: функциональные сервисы вызывают друг друга, как показано на рис. 2.3.

У такого подхода есть преимущество: клиенты получаются простыми и даже асинхронными. Клиент выдает вызов сервиса A; затем сервис A вызывает B, а B вызывает C.

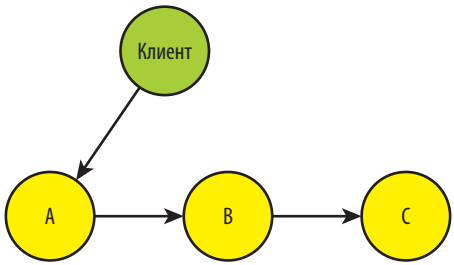


Рис. 2.3. Цепочки функциональных сервисов

Но теперь появляется другая проблема: функциональные сервисы тесно связаны друг с другом и с порядком функциональных вызовов. Например, сервис оплаты может вызываться только после сервиса выставления счета, но до сервиса отгрузки. В случае рис. 2.3 в сервис A встраивается информация о том, что он должен вызвать сервис B. Сервис B может вызываться только после сервиса A и до сервиса C. Изменение в требуемом порядке вызовов с большой вероятностью отразится на всех сервисах выше и ниже в цепочке, потому что их реализация должна измениться в соответствии с новыми требованиями к порядку.

Но рис. 2.3 не раскрывает полной картины. Сервис B на рис. 2.3 кардинально отличается от сервиса B на рис. 2.1. Исходный сервис B выполнял только функциональность B. Сервис B на рис. 2.3 должен знать о существовании

сервиса C, а в контракт B должны быть включены параметры, которые потребуются сервису C для выполнения его функциональности. На рис. 2.1 за эти подробности отвечал клиент.

Проблема усугубляется сервисом A. Теперь он должен включить в свой контракт параметры, необходимые для вызова сервисов B и C, чтобы те могли выполнить свою соответствующую бизнес-функциональность. Любые изменения в функциональности B и C отражаются в изменениях реализации сервиса A, который теперь тесно связан с ними. Разбухание и привязки такого рода представлены на рис. 2.4.

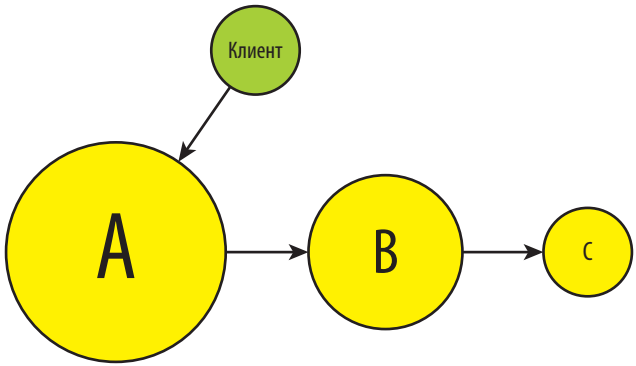


Рис. 2.4. Объединение функциональности в цепочки приводит к разбуханию сервисов

Как ни печально, даже рис. 2.4 не показывает всей правды. Допустим, сервис A успешно выполнил функциональность A, после чего перешел к вызову сервиса B для выполнения функциональности B. Однако сервис B столкнулся с ошибкой и не смог нормально отработать. Если сервис A вызывал B синхронно, то он должен располагать полной информацией о внутренней логике и состоянии B, чтобы провести восстановление после ошибки. А это означает, что функциональность B также должна присутствовать в сервисе A. Если A вызывает B асинхронно, то сервис B должен каким-то образом вернуть информацию A для отмены функциональности A или же выполнить откат A собственными силами. Другими словами, функциональность A также должна присутствовать в B. Тем самым создается тесная связь между сервисами B и A, а сервис B разбухает из-за необходимости компенсировать успешное выполнение сервиса A. Ситуация изображена на рис. 2.5.

Проблема усугубляется сервисом C. Что, если функциональности A и B отработали успешно и завершились, но сервис C не смог выполнить свою бизнес-функцию? Сервис C должен вернуться к сервисам B и A для отмены их операций. Это приводит к дополнительному разбуханию сервиса C и его связыванию с сервисами A и B. Если взять ситуацию на рис. 2.5, что потребуется для

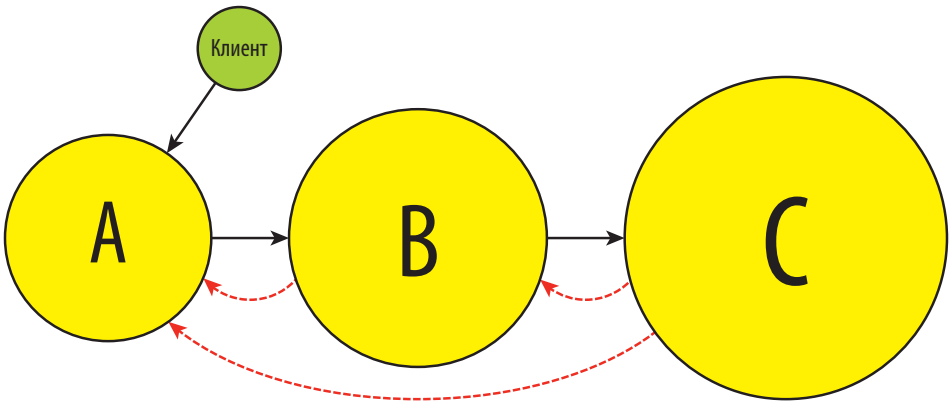


Рис. 2.5. Дополнительное разбухание и связи из-за необходимости компенсации

замены сервиса B сервисом B', который выполняет свою функциональность не так, как B? Какие нежелательные последствия это будет иметь для сервисов A и C? И какая степень повторного использования достигается на рис. 2.5, если функциональность сервисов будет востребована в других контекстах — скажем, если сервис B вызывается после D, но до E? Что собой представляют A, B и C — три разнородных сервиса или одну спекшуюся бесформенную массу?

Относительно функциональной декомпозиции

В функциональной декомпозиции кроется почти неудержимый соблазн. Она кажется простым и логичным способом проектирования системы: от вас требуется всего лишь составить список необходимых функциональностей, а затем создать в архитектуре компонент для каждого пункта. Функциональная декомпозиция (и ее близкий родственник — декомпозиция предметных областей, о которой будет рассказано ниже) применяется при проектировании большинства систем. Многие проектировщики считают функциональную декомпозицию естественным выбором, и скорее всего, ваш преподаватель информатики в институте объяснял вам именно этот способ. Распространенность функциональной декомпозиции в плохо спроектированных системах — почти идеальный признак того, что от нее стоит держаться подальше. Всеми силами сопротивляйтесь искушениям функциональной декомпозиции.

О природе Вселенной (TANSTAAFL)

Чтобы доказать, что функциональная декомпозиция нежизнеспособна, можно вообще обойтись без аргументов из области программирования. Доказательство кроется в самой природе Вселенной, а конкретно в первом законе термодинамики. Если отбросить математику, первый закон термодинамики просто

гласит, что нельзя создать что-то полезное без приложения усилий. В просторечии это называется принципом TANSTAAFL (сокращение от «There ain't no such thing as a free lunch», то есть «бесплатных завтраков не бывает» — в смысле «даром ничего не дается»).

Проектирование по своей природе является деятельностью с высокой добавочной ценностью. Вы читаете эту книгу вместо еще одной книги по программированию именно потому, что вы понимаете полезность проектирования, или, говоря иначе, вы считаете, что проектирование обладает добавочной ценностью (и немалой!).

Проблема функциональной декомпозиции заключается в том, что она пытается обойти первый закон термодинамики. Результат функциональной декомпозиции (а именно проектирование системы) должен быть деятельностью с высокой добавочной ценностью. Однако функциональная декомпозиция проста и прямолинейна: для заданного набора требований, предусматривающих выполнение функциональностей A, B и C, вы проводите декомпозицию на сервисы A, B и C. «Проще простого! — говорите вы. — Функциональная декомпозиция настолько проста, что с ней справится даже программа». Но именно потому, что этот способ проектирования отличается быстротой, простотой, механистичностью и прямолинейностью, он противоречит первому закону термодинамики. Так как ценность не может добавляться без усилий, сами атрибуты, которые делают функциональную декомпозицию столь привлекательной, не позволяют ей создавать добавочную ценность.

Антипроектирование

Убедить коллег и руководство использовать что-то другое вместо функциональной декомпозиции — весьма непростое дело. «Мы всегда так делали», — скажут они вам. На этот довод можно привести два возражения. Во-первых, можно ответить: «И сколько раз мы выдерживали сроки или бюджет, которые были утверждены изначально? Что у нас получалось с качеством и сложностью? Насколько просто осуществлялось сопровождение системы?»

Во-вторых, можно провести сеанс антипроектирования. Сообщите команде, что вы проводите конкурс по проектированию системы следующего поколения. Разбейте команду на две половины и рассадите их в разных комнатах. Предложите первой половине разработать лучшее проектировочное решение для системы. Затем предложите второй половине разработать худшее из всех возможных решений: то, которое бы предельно затрудняло сопровождение и расширение системы, которое бы препятствовало повторному использованию и т. д. Дайте им поработать в течение нескольких часов, а затем соберите вместе. При сравнении результатов обычно выясняется, что они выдали практически одинаковые результаты. Подписи на компонентах могут различаться, но суть останется неизменной. Только теперь признайтесь, что они работали

над разными задачами, и обсудите последствия. Возможно, пришло время поискать новый подход.

Пример: функциональный дом

Тот факт, что при проектировании никогда не следует применять функциональную декомпозицию, — универсальное наблюдение, которое не имеет никакого отношения к программным системам. Возьмем построение функциональности дома, как если бы он был программной системой. Все начинается с перечисления необходимой функциональности: приготовление еды, игры, отдых, сон и т. д. Затем для каждого вида функциональности в архитектуре создается отдельный компонент, как показано на рис. 2.6.

Рис. 2.6. Функциональная декомпозиция дома

Хотя рис. 2.6 уже выглядит нелепо, настоящее безумие проявляется только тогда, когда приходит время строить дом. Вы начинаете с пустого участка земли и беретесь за приготовление еды. Только приготовление еды. Вы вынимаете микроволновку из коробки и откладываете ее в сторону. Затем вы заливаете бетоном маленькую площадку, ставите на нее деревянный каркас, накрываете крышкой и ставите на нее микроволновку. Остается построить для микроволновки маленький чулан, сколотить над ней крошечную крышу и подключить к электросети. «Порядок, теперь можно готовить!» — объявляете вы начальству и заказчикам.

Но так ли это? Разве можно готовить еду таким способом? Где вы собираетесь ее подавать, где хранить остатки, куда выкидывать мусор? А как насчет приготовления еды на газовой плите? Что потребуется для того, чтобы повторить эту процедуру для плиты? Какая степень повторного использования может быть достигнута между этими двумя разными способами выражения функциональности приготовления еды? Можно ли легко расширить только одну

из них? А если вам потребуется переставить микроволновку в другое место? Все эти проблемы даже не могут считаться первым шагом, потому что все зависит от того, что именно вы готовите. Возможно, вам придется строить разную функциональность приготовления еды, если в процессе приготовления задействованы разные кухонные устройства, а результаты могут отличаться по контексту — например, если вы готовите завтрак, обед, ужин, десерт или легкую закуску. В результате вы получаете либо мириады крошечных сервисов, предназначенных для конкретного сценария, который должен быть известен заранее, либо получается один огромный сервис, в котором есть все. Возможно ли построить дом при подобном подходе? А если нет, то стоит ли так проектировать и строить программные системы?

КОГДА ПРИМЕНЯЕТСЯ ФУНКЦИОНАЛЬНАЯ ДЕКОМПОЗИЦИЯ

Все это глумление вовсе не означает, что функциональная декомпозиция в принципе плоха. У функциональной декомпозиции есть свое место — это превосходный метод выявления требований. Она помогает архитекторам (или руководителям продукта) выявлять скрытые или неочевидные области функциональности. Даже при нечетких функциональных требованиях можно начать с верхнего уровня и провести функциональную декомпозицию до уровня с высокой детализацией. Вы выявляете требования и отношения между ними, встраиваете требования в древовидную систему и выявляете избыточные аспекты или взаимоисключающую функциональность. Тем не менее попытка превратить функциональную декомпозицию в проектировочное решение ведет к фатальным последствиям. Между требованиями и проектировочным решением никогда не должно быть прямого соответствия.

Недостатки декомпозиции предметной области

Проектное решение дома на рис. 2.6 очевидно абсурдно. Скорее всего, в своем доме вы предпочитаете готовить на кухне, поэтому на рис. 2.7 изображена альтернативная декомпозиция дома. Такая форма декомпозиции называется *декомпозицией предметной области*: система разбивается на структурные элементы в соответствии с предметными бизнес-областями: продажи, техническое обеспечение, бухгалтерия, отгрузка и т. д. К сожалению, декомпозиция предметной области (рис. 2.7) на практике работает еще хуже функциональной декомпозиции на рис. 2.6. Причина заключается в том, что она остается замаскированной функциональной декомпозицией: на кухне (`Kitchen`) вы готовите, в спальне (`Bedroom`) спите, в гараже (`Garage`) ставите машину и т. д.

Собственно, каждая из функциональных областей на рис. 2.6 может быть отображена на предметные области на рис. 2.7, что создает серьезные проблемы.

Каждая спальня может быть уникальной, но функциональность сна должна дублироваться во всех спальнях. Дальнейшее дублирование возникает тогда, когда вы спите перед телевизором в гостиной или развлекаете гостей на кухне (практически все вечеринки почему-то заканчиваются на кухне).

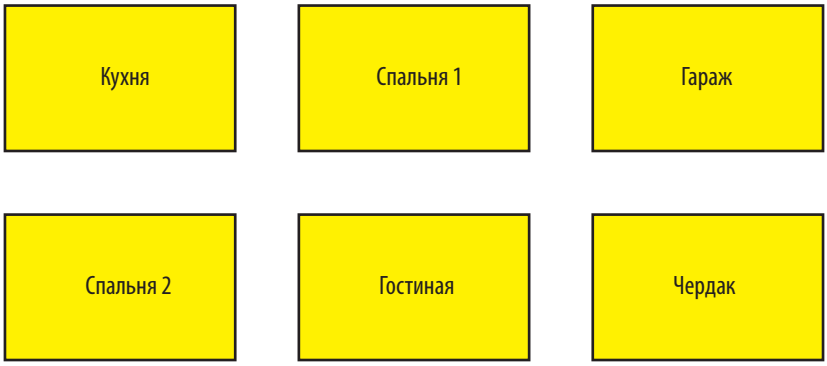


Рис. 2.7. Декомпозиция предметной области для дома

Каждая предметная область часто деградирует до уродливой свалки функциональности, что только повышает ее внутреннюю сложность. Повышение внутренней сложности заставляет разработчика держаться подальше от проблем межобластных связей, а взаимодействие между предметными областями обычно сокращается до простых изменений состояния (в стиле CRUD) вместо действий, инициирующих выполнение нужного поведения во всех областях. Компоновка более сложных вариантов поведения между предметными областями становится очень сложной задачей. Некоторые функциональности просто не могут быть представлены в декомпозициях предметных областей. Для примера возьмем дом на рис. 2.7: где бы вы готовили еду, которая не может готовиться на кухне (например, барбекю)?

Построение дома с декомпозицией предметной области

Как и с чисто функциональным подходом, настоящие проблемы с декомпозицией предметной области становятся очевидными в ходе построения. Представьте, что вы строите дом на основе декомпозиции на рис. 2.7. Строительство начинается с пустого участка земли. Вы роете котлован для фундамента кухни, заливаете его бетоном (только для кухни!) и добавляете арматуру. Затем вы воздвигаете стены кухни (все они должны быть внешними), закрепляете их на фундаменте, прокладываете электропроводку и трубы в стенах; подключаете кухонное оборудование к водопроводу, газопроводу и электросети; устанавливаете системы обогрева и охлаждения; ставите счетчики расхода воды, электричества и газа; строите крышу над кухней; красите стены изнутри; раз-

вешиваете шкафы; покрываете штукатуркой наружные стены (то есть все стены) и красите их. Наконец, вы объявляете заказчику, что модуль `Kitchen` готов, а контрольная точка 1.0 достигнута.

Затем вы переходите к спальне. Сначала вы сбиваете штукатурку с кухонных стен, чтобы обнажить крепления стен, и отсоединяете кухню от фундамента. Вы отключаете кухню от газопровода, водопровода и электросети и канализации, а затем при помощи дорогостоящих гидравлических домкратов поднимаете ее. Пока кухня висит в воздухе, вы сдвигаете ее в сторону, чтобы вам было удобнее разломать фундамент отбойными молотками, выкинуть мусор и оплатить ее вывоз. Теперь можно вырыть новый котлован под общий фундамент для спальни и кухни. Вы заливаете котлован бетоном и устанавливаете новые крепления, стараясь разместить их точно в тех местах, что и прежде. Затем вы очень осторожно опускаете кухню на новый фундамент и убеждаетесь в том, что все крепления и отверстия совпадают (что почти невозможно). Вы строите новые стены для спальни. Потом вы временно снимаете шкафы с кухонных стен, сбиваете штукатурку, чтобы получить доступ к вмонтированным проводам и трубам, и соединяете все вентиляционные каналы, трубы и провода с каналами, трубами и проводами в спальне. Вы снова покрываете штукатуркой стены в кухне и спальне, вешаете обратно шкафы и расставляете мебель в спальне. Всю оставшуюся на стенах кухни штукатурку приходится сбить, чтобы наложить сплошную штукатурку без трещин на внешние стены. Некоторые стены кухни, которые прежде были внешними, теперь стали внутренними — со всеми последствиями для штукатурки, изоляции, краски и т. д. Остается снять крышу с кухни и возвести новую общую крышу для кухни и спальни. Теперь можно сообщить заказчику, что контрольная точка 2.0 достигнута, а модуль `Bedroom 1` готов.

О том факте, что вам пришлось перестраивать кухню, никто вслух не говорит — как и о том, что повторное строительство кухни обошлось намного дороже и было сопряжено с большим риском, чем первое. А если вам потребуется добавить к этому дому еще одну спальню? Сколько раз вы будете строить и ломать кухню? Сколько раз вообще можно перестроить кухню, прежде чем она рассыплется в передвигаемую с места на место груду бесполезного мусора? Была ли кухня действительно готова в тот момент, когда вы об этом объявили? Даже если забыть о потерях на перестройку, в какой степени возможно повторное использование различных частей дома? Насколько дороже обойдется строительство при таком подходе? И стоит ли применять такой подход для построения программной системы?

Ошибочная мотивация

Стандартный довод для применения функциональной декомпозиции или декомпозиции предметной области — заказчик хочет получить ту или иную

функцию как можно быстрее. Проблема в том, что одна функция никогда не может быть реализована в изоляции от других. Подсистема оплаты не обладает бизнес-ценностью независимо от подсистем выставления счетов и отгрузки.

С унаследованными системами ситуация становится еще хуже. Разработчикам редко достается такая роскошь, как построение совершенно новой системы на пустом месте. Гораздо вероятнее, что им достанется существующая разваливающаяся система, которая была спроектирована на основе функциональной декомпозиции; негибкость и высокие затраты на сопровождение оправдывают разработку новой системы.

Предположим, ваш бизнес использует три функциональности, A, B и C, работающие в унаследованной системе. Когда вы строите новую систему для замены старой, вы решаете построить — и что еще важнее, развернуть — функциональность A в первую очередь, чтобы удовлетворить заказчиков и руководителей, которые желают видеть создаваемую ценность как можно раньше и чаще. Проблема в том, что бизнесу не нужна функциональность A сама по себе — только в сочетании с B и C. Если A выполняется в новой системе, а B и C — в старой, решение работать не будет, потому что старая система не знает о новой и не может выполнять только B и C. Выполнение A как в старой, так и в новой системе не создает никакой ценности и даже сокращает ценность из-за дублирования работы, так что, скорее всего, пользователям это не понравится. Единственный выход — как-то согласовать новую систему со старой. Обычно такое согласование по сложности намного превосходит исходную бизнес-задачу, так что в итоге разработчикам придется решать намного более сложную задачу. Возвращаясь к аналогии с домом, что бы вы сказали, если бы вам пришлось жить в тесном старом доме, пока на другой стороне города строится новый дом по плану на рис. 2.6 или 2.7? Допустим, вы строите только подсистему приготовления еды или кухню в новом доме, но при этом продолжаете жить в старом. Каждый раз, когда вы проголодаетесь, вам придется ехать в новый дом и возвращаться обратно. Со своим домом вы бы на такое не согласились, и навязывать подобные неудобства заказчикам не следует.

Тестируемость и проектирование

Критический недостаток как функциональной декомпозиции, так и декомпозиции предметной области связан с тестированием. В таких архитектурах количество связей и сложность настолько высоки, что единственный вид тестирования, на который могут пойти разработчики, — это модульное тестирование. Впрочем, это не показатель важности модульного тестирования, а всего лишь еще один пример «эффекта уличного фонаря»[1] (когда вы ищете не там, где потеряли, а где светлее и где легче найти).

[1] https://en.wikipedia.org/wiki/Streetlight_effect

ФИЗИЧЕСКИЕ И ПРОГРАММНЫЕ СИСТЕМЫ

В этой книге я использую примеры из материального мира (например, дома) для демонстрации общих принципов проектирования. В отрасли разработки часто встречается мнение, что принципы проектирования физических сущностей нельзя экстраполировать на программы, что процессы проектирования и построения программ каким-то образом избавлены от ограничений на проектирование или процессы физических систем. В конце концов, в программе можно сначала нарисовать дом, а затем построить стены «под краску». Вам не нужно возиться с такими подробностями, как балки и кирпичи.

Я обнаружил, что отрасль не только может заимствовать опыт и передовые приемы из материального мира, но и обязана это делать. Вопреки интуитивным представлениям, для программ проектирование еще более необходимо, чем для физических систем. Причина в сложности. Сложность физических систем, таких как типичный дом, должна учитывать физические ограничения. Невозможно построить плохо спроектированный дом с сотнями взаимосвязанных коридоров и комнат. Стены окажутся слишком тяжелыми, будут содержать слишком много проемов, окажутся слишком тонкими, двери будут слишком маленькими или проект окажется слишком дорогостоящим для реализации. Невозможно использовать слишком много строительных материалов, потому что дом обрушится, вам не хватит средств на их покупку или места для хранения запасов.

Без таких естественных физических ограничений сложность программных систем может быстро выйти из-под контроля. Обуздать эту сложность можно только одним способом: применением хороших инженерных методов, для которых проектирование и технологические процессы играют основополагающую роль. Хорошо спроектированные программные системы очень похожи на физические сущности и строятся практически по тем же правилам. Они напоминают хорошо спроектированные механизмы.

Функциональная декомпозиция или декомпозиция предметной области не имеет смысла при проектировании и построении домов или программных систем. Все сложные сущности (физические или нет) обладают одинаковыми абстрактными атрибутами, от дерева решений проектирования до критического пути выполнения. Все составные системы должны проектироваться так, чтобы они были безопасными, простыми в сопровождении, пригодными для повторного использования, расширяемыми и высококачественными. Это утверждение относится как к домам, так и к частям машин или программным системам. Все это непременные атрибуты практической инженерной деятельности, и реализовать и поддерживать их можно только одним способом: применением универсальных инженерных практик.

Несмотря на все сказанное, между физическими и программными системами существует принципиальное различие: видимость. Любой, кто попытается построить дом на рис. 2.6 или 2.7, будет немедленно уволен. Такой человек явно неадекватен, а ужасающие потери строительных материалов, времени и средств, а также высокий риск травматизма будут очевидны всем окружающим. С программными системами дело обстоит иначе: колоссальные потери тоже присутствуют, но они скрыты. В программировании пыль и строительный мусор заменяются загубленными карьерными перспективами, энергией и молодостью. Тем не менее никто никогда не видел и не обращал внимания на эти скрытые потери, а безумие не только разрешается, но и поощряется, словно психушка перешла под контроль пациентов. Правильное проектирование позволит вам освободиться и вернуть контроль над происходящим за счет исключения этих скрытых потерь. Как будет показано в части II книги, к планированию проектов это относится еще в большей мере.

Как ни печально, модульное тестирование находится на грани бесполезности. Хотя модульное тестирование является неотъемлемой частью тестирования, оно не способно протестировать систему. Представьте авиалайнер с множеством внутренних компонентов (насосы, приводы, сервомеханизмы, турбины и т. д). Теперь предположим, что все компоненты по отдельности идеально прошли модульное тестирование, но все тестирование выполнялось только до сборки самолета из компонентов. Рискнули бы вы сесть в такой самолет? Причина, по которой модульное тестирование имеет крайне ограниченную ценность, состоит в том, что в любой сложной системе дефекты не скрываются в модулях, а возникают в результате взаимодействия между модулями. Вот почему вы инстинктивно понимаете, что каждый компонент лайнера может отлично работать, но результат их сборки может быть абсолютно ошибочным. Что еще хуже, даже если сложная система пребывает в идеальном состоянии с безупречным качеством, изменение одного компонента, прошедшего модульное тестирование, может нарушить работу другого модуля (или модулей), зависящего от старого поведения. При изменении всего одного модуля приходится повторять тестирование всех модулей. Но даже тогда это будет бессмысленно, потому что изменение одного из компонентов может повлиять на взаимодействия между другими компонентами или подсистемами, которые не сможет выявить никакое модульное тестирование.

Единственный способ проверки изменений — полное регрессионное тестирование системы, ее подсистем, ее компонентов и взаимодействий и, наконец, ее модулей. Если в результате изменения придется вносить изменения в другие модули, то его эффект для регрессионного тестирования будет нелинейным. Идея о неэффективности модульного тестирования далеко не нова, она

была продемонстрирована на тысячах систем с вычислением объективных метрик.

В теории регрессионное тестирование можно применять даже в системе с функциональной декомпозицией. На практике сложность этой задачи устанавливает очень высокую планку. Из-за гигантского количества функциональных компонентов тестирование всех взаимодействий становится практически нецелесообразным. Внутреннее устройство очень больших сервисов окажется настолько сложным, что никто не сможет эффективно разработать полноценную стратегию для тестирования всех ветвей выполнения таких сервисов. С функциональной декомпозицией многие разработчики обычно сдаются и ограничиваются простым модульным тестированием. Таким образом, функциональная декомпозиция, исключающая возможность регрессионного тестирования, делает всю систему нетестируемой, а нетестируемые системы всегда изобилуют дефектами.

Пример: функциональная торговая система

Вместо дома рассмотрим следующие упрощенные требования к системе биржевой торговли для финансовой компании:

- Система должна предоставлять внутренним брокерам следующие возможности:
  - покупка и продажа акций;
  - планирование сделок;
  - построение отчетов;
  - анализ результатов.
- Пользователи системы используют браузер для подключения к системе и управления сеансами, заполнения форм и отправки запросов.
- После запроса на проведение сделки, построение отчета или проведение анализа система отправляет пользователям сообщения электронной почты с подтверждением или результатами запроса.
- Данные должны храниться в локальной базе данных.

Прямолинейная функциональная декомпозиция дает результат, показанный на рис. 2.8.

Каждое из функциональных требований выражается соответствующим компонентом архитектуры. На рис. 2.8 представлено обобщенное проектировочное решение, за которое многие начинающие разработчики возьмутся без малейших сомнений.

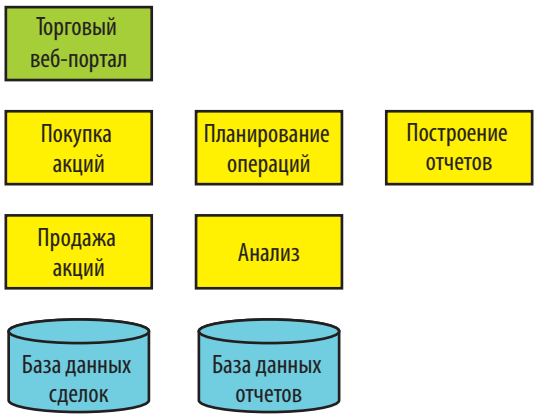


Рис. 2.8. Функциональная торговая система

Проблемы функциональной торговой системы

Такое проектировочное решение имеет множество недостатков. Весьма вероятно, что клиент в представленной системе будет координировать покупку акций, продажу акций и планирование операций; выдавать запросы на построение отчетов и т. д. Предположим, пользователь хочет профинансировать покупку определенного количества акций за счет продажи других акций. Это подразумевает два запроса: сначала на продажу, затем на покупку. Но что делать клиенту, если к моменту выполнения этих двух транзакций цена продаваемых акций упала или цена покупаемых акций поднялась так, что продажа не сможет покрыть покупку? Должен ли клиент купить столько акций, сколько сможет? Или продать больше акций, чем предполагалось изначально? Может ли он заимствовать средства из расходного счета, связанного с учетной записью продавца, для финансирования заказа? А может, отменить всю сделку? Или обратиться за помощью к пользователю? Точная процедура разрешения проблемы несущественна для этого обсуждения. Какой бы способ ни был выбран, он потребует бизнес-логики, которая теперь находится в клиенте.

А что потребуется для того, чтобы клиент смог работать с системой не через веб-портал, а из приложения на мобильном устройстве? Не приведет ли это к дублированию бизнес-логики на мобильном устройстве? Скорее всего, бизнес-логику и усилия, затраченные на разработку веб-клиента, практически не удастся повторно использовать в мобильном клиенте, потому что они встроены в веб-портал. Со временем все придет к тому, что разработчики начнут поддерживать несколько версий бизнес-логики в разных клиентах.

Что касается требований, подсистемы Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ уведомляют пользователя

о своей деятельности по электронной почте. А если пользователь предпочитает получать текстовые сообщения (или бумажные письма) вместо электронной почты? Придется изменять реализацию подсистем Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ и переделывать электронную почту в текстовые сообщения.

В соответствии с проектировочным решением информация хранится в базе данных, а подсистемы Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ обращаются к базе данных. Теперь предположим, что вы решите переместить данные из локальной базы данных в облачное решение. Как минимум это заставит вас изменить код обращения к данным в подсистемах Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ. Структура, процессы обращения к данным и их потребления должны быть изменены во всех компонентах.

Что, если клиент захочет взаимодействовать с системой асинхронно — выдать несколько команд на выполнение операций и получить результаты позднее? Компоненты были построены в предположении о подключенном синхронном клиенте, который координирует работу компонентов. Скорее всего, вам придется переписать функциональность Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ, чтобы они координировали друг друга в соответствии с рис. 2.5.

Финансовые портфели часто содержат другие финансовые инструменты помимо акций: валюты, облигации, товары и даже опционы и фьючерсы. Что, если пользователи системы захотят начать торговать валютой или товарами вместо акций? Что, если пользователи потребуют единое приложение для управления всеми своими портфелями (вместо нескольких приложений)? Подсистемы Покупка акций, Продажа акций и Планирование операций ориентированы на работу с акциями, они не поддерживают операции с валютой или облигациями, из-за чего вам придется добавлять дополнительные компоненты (как на рис. 2.6). Аналогичным образом придется переделать модули Построение отчетов и Анализ для того, чтобы они поддерживали отчеты и анализ по другим операциям, кроме акций. Клиент тоже придется переписать для поддержки новых видов операций.

Впрочем, даже без расширения видов деятельности — что, если приложение потребуется локализовать для зарубежных рынков? Как минимум клиент нужно будет серьезно переработать для поддержки других языков, но настоящие последствия снова проявятся в компонентах системы. На зарубежных рынках будут действовать другие правила торговли, законодательные акты и требования, которые кардинально повлияют на то, что может делать система и как должны проводиться операции. Это будет означать значительную переработку подсистем Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ каждый раз, когда в системе появляется новый локальный контекст. В итоге у вас получатся либо разбухшие всемогущие сер-

висы, способные торговать на любом рынке, либо разные версии систем для всех локальных контекстов развертывания.

Наконец, все компоненты в настоящее время подключаются к каналу поставки биржевых котировок, который дает новейшую информацию о ценах акций. Что потребуется для перехода на новый канал поставки данных или поддержки разных каналов? Как минимум подсистемы Покупка акций, Продажа акций, Планирование операций, Построение отчетов и Анализ потребуют определенной работы для перехода на новые каналы, подключение к ним, обработку ошибок, оплату услуг и т. д. Нет никаких гарантий того, что новый канал использует такой же формат данных, как и старый. Все компоненты также потребуют работы по преобразованию и переработке.

Декомпозиция на основе нестабильности

Основное правило проектирования Метода гласит:

Выполняйте декомпозицию на основе нестабильности.

Декомпозиция на основе нестабильности (*Volatility-based decomposition*) выявляет области потенциальных изменений и инкапсулирует их в сервисах или структурных элементах системы. Затем требуемое поведение реализуется в форме взаимодействий между инкапсулированными областями нестабильности. Основным доводом в пользу декомпозиции на основе нестабильности является простота: любые изменения инкапсулируются, а их воздействие на систему изолируется.

При применении декомпозиции на основе нестабильности вы начинаете рассматривать систему в виде последовательности хранилищ, как показано на рис. 2.9.

Любые изменения содержат потенциальную опасность, как ручная граната с выдернутой чекой. С декомпозицией на основе нестабильности вы открываете дверь соответствующего хранилища, бросаете гранату и закрываете дверь. Возможно, все содержимое хранилища будет полностью уничтожено, но за его пределами не будет осколков, уничтожающих все на своем пути. Вам удалось сдержать изменения.

С функциональной декомпозицией ваши структурные элементы представляют области функциональности, а не области нестабильности. В результате при возникновении изменений по самому определению декомпозиции они затронут многие (если не все) компоненты вашей архитектуры. Следовательно, функциональная декомпозиция максимизирует эффект изменений. Так как большинство программных систем проектируется по функциональному принципу, изменения часто оказываются болезненными и дорогостоящими, а эффект изменений с большой вероятностью вызовет резонанс во всей системе.

Изменения, внесенные в одной области функциональности, инициируют другие изменения и т. д. Контроль изменений — настоящая причина, по которой следует избегать функциональной декомпозиции.

Рис. 2.9. Инкапсуляция областей нестабильности
(изображения: media500/Shutterstock; pikepicture/Shutterstock)

Все остальные проблемы функциональной декомпозиции уходят на второй план по сравнению с ограниченными возможностями и высокими затратами на сдерживание изменений. С функциональной декомпозицией изменение производят такой же эффект, как проглоченная взведенная граната.

То, что вы решите инкапсулировать, может иметь функциональную природу, но вряд ли когда-либо сочетает функциональность с предметной областью, а следовательно, не имеет смысла с точки зрения бизнеса. Например, электричество, питающее дом, действительно является областью функциональности, но также является важной областью для инкапсуляции по двум причинам. Первая причина заключается в том, что электропитание обладает высокой нестабильностью: ток может быть постоянным или переменным; с напряжением 110 или 220 вольт; однофазным или трехфазным; с частотой 50 или 60 Гц; он может поступать от солнечных панелей на крыше, с генератора на заднем дворе или от простого подключения к электросети; поступать по проводам с разными характеристиками и т. д. и т. п. Вся эта нестабильность инкапсулируется за розеткой. Когда приходит момент потребления электроэнергии, все, что видит пользователь, — это розетка, за которой инкапсулирована вся нестабильность. Таким образом устройства, потребляющие энергию, изолируются от нестабильности, что улучшает возможности повторного использования, безопасность и расширяемость системы при сокращении общей сложности. В результате использование электропитания в одном доме не отличается от его

использования в другом доме; так мы приходим ко второй причине, по которой питание нужно определить как то, что стоит инкапсулировать в доме. Хотя энергопитание в доме является областью функциональности, в общем случае использование питания не привязано ни к какой предметной области (члены семьи, живущие в доме, их отношения, их благополучие и т. д).

С чем можно сравнить жизнь в доме, в котором нестабильность питания не была инкапсулирована? Каждый раз, когда вы захотите воспользоваться электроэнергией, вам придется сначала добраться до проводов, измерить частоту на осциллографе и проверить напряжение вольтметром. И хотя электричеством можно пользоваться и так, намного проще положиться на инкапсуляцию всей нестабильности за розеткой. Это позволит вам создавать новую ценность с интеграцией электропитания в свои задачи или процессы.

Декомпозиция, сопровождение и разработка

Как объяснялось ранее, функциональная декомпозиция радикально увеличивает сложность системы. Функциональная декомпозиция также превращает сопровождение в сущий кошмар. Мало того что код таких систем получается слишком сложным, изменения распределяются между разными сервисами. В результате сопровождение кода становится слишком трудоемким, создает повышенный риск ошибок и занимает очень много времени. В общем случае чем сложнее код, тем ниже его качество, а низкое качество создает еще больше проблем с сопровождением. Вам приходится преодолевать высокую сложность и избегать внесения новых дефектов при устранении старых. В системе с функциональной декомпозицией новые изменения часто порождают новые дефекты из-за слияния низкого качества и сложности. При расширении функциональной системы затраты часто оказываются непропорционально высокими в отношении выгоды для заказчика.

Функциональная декомпозиция создает риски еще до того, как начнется сопровождение, — пока система находится в процессе разработки. Требования изменяются в ходе разработки (это происходит всегда); затраты на каждое изменение оказываются огромными, отражаются на нескольких областях, требуют существенных переделок и в конечном итоге ставят под угрозу график сдачи продукта.

Системы, спроектированные с использованием декомпозиции на основе нестабильности, резко отличаются от них своей способностью реагировать на изменения. Так как изменения содержатся в каждом модуле, по крайней мере появляется надежда на простое сопровождение без побочных эффектов за границами модуля. Сокращение сложности и упрощение сопровождения приводит к значительному повышению качества. Если что-то инкапсулировано точно так же, как в другой системе, открывается возможность повторного использования. Систему можно расширить, добавляя к ней новые области инкапсулированной нестабильности или интегрируя существующие области нестабильности други-

ми способами. Инкапсуляция нестабильности означает гораздо большую устойчивость перед явлением «ползучего улучшизма» и позволяет выдержать сроки, так как изменения будут иметь ограниченные последствия.

Универсальный принцип

Преимущества декомпозиции на основе нестабильности проявляются не только в программных системах. Они являются универсальными принципами хорошего проектирования во многих областях: от финансов до бизнеса, от биологии до физических систем и качественных программных продуктов. Универсальные принципы по своей природе действуют и для программных систем (иначе они не были бы универсальными). Для примера рассмотрим ваше собственное тело. Функциональная декомпозиция выделила бы компоненты для каждого вида деятельности, которую вы обязаны выполнять, от управления машиной до программирования, но при этом в вашем теле нет соответствующих компонентов. Такие задачи, как программирование, решаются посредством интеграции областей нестабильности.

Например, сердце предоставляет важный сервис для вашей системы: оно прокачивает кровь. Прокачка крови сопряжена с огромной нестабильностью: высокое и низкое давление, содержание соли, вязкость, частота пульса, вид деятельности (вы сидите на месте или бежите), с адреналином и без, разные группы крови, здоровое и болезненное состояние организма и т. д. Тем не менее вся эта нестабильность инкапсулируется за сервисом, который называется «сердцем». Сможете ли вы программировать, если вам придется постоянно беспокоиться о нестабильности, связанной с прокачкой крови?

> **ПРИМЕЧАНИЕ** Природа, работающая в условиях почти бесконечного времени и запаса энергии при почти 0% эффективности, пришла к декомпозиции на основе нестабильности. Тем не менее в мире людей ресурсы более ограниченны. Люди могут пользоваться преимуществами опробованных инженерных принципов, творческого интеллекта и способностью передавать знания, которые помогают избежать неизбежных тупиков, возникающих при применении метода проб и ошибок. Декомпозиция на основе нестабильности — высшее проявление технических процессов, строящихся на базе принципов природы.

Вы также можете интегрировать в свою реализацию внешние области инкапсулированной нестабильности. Взгляните на свой компьютер: он отличается буквально от каждого компьютера в этом мире, однако вся эта нестабильность инкапсулирована. Пока компьютер может отправить сигнал на экран, вас не интересует, что происходит за графическим портом. Вы решаете задачи программирования, интегрируя инкапсулированные области нестабильности — как внешние, так и внутренние. Одни и те же области нестабильности (например, сердце) могут использоваться повторно для выполнения других видов

функциональности, таких как управление машиной или представление работы заказчику. Другого способа проектирования и построения жизнеспособной системы попросту не существует.

Декомпозиция на основе нестабильности — сущность проектирования системы. Все хорошо спроектированные системы, как программные, так и физические, инкапсулируют свою нестабильность внутри структурных элементов системы.

Декомпозиция на основе нестабильности и тестирование

Декомпозиция на основе нестабильности хорошо сочетается с регрессионным тестированием. Сокращение количества компонентов, сокращение размера компонентов и упрощение взаимодействий между компонентами — все эти факторы кардинально сокращают сложность системы. Это позволяет организовать регрессионное тестирование, при котором система тестируется в полном объеме, каждая подсистема тестируется по отдельности, а в конечном итоге также тестируются независимые компоненты. Поскольку декомпозиция на основе нестабильности ограничивает изменения в рамках структурных элементов системы, даже неизбежно возникающие изменения не препятствуют регрессионному тестированию. Эффект изменения компонента можно протестировать в изоляции от остальных частей системы без ущерба для тестирования взаимодействий между другими компонентами и подсистемами.

НА ПЛЕЧАХ ГИГАНТОВ: ДЭВИД ПАРНАС

В 1972 году Дэвид Парнас (David Parnas), один из основоположников теории программирования, опубликовал статью «О критериях, применяемых при декомпозиции систем на модули».[1]

В этой короткой статье, состоявшей всего из 5 страниц, содержится большинство элементов современной технологии программирования, включая инкапсуляцию, сокрытие информации, сцепление, модули и слабые связи. И что самое важное, Парнас обозначил в этой статье необходимость выбора изменений в качестве ключевого критерия декомпозиции (вместо функциональности). Хотя конкретное содержимое статьи сейчас выглядит архаично, это был первый случай, когда кто-то в отрасли задал актуальные вопросы о том, что необходимо сделать для того, чтобы программные системы были удобными в сопровождении, пригодными для повторного использования и расширяемыми. Как следствие, эта статья стала точкой зарождения современной технологии программирования. Следующие 40 лет Парнас старался внедрить проверенные классические практики в области разработки программного обеспечения.

[1] Communications of the ACM 15, no. 12 (1972): 1053–1058.

Проблемы нестабильности

Идеи и мотивация декомпозиции на основе нестабильности просты и практичны, они хорошо соответствуют реальности и здравому смыслу. Основные проблемы при выполнении декомпозиции на основе нестабильности связаны со временем, общением и восприятием. Оказывается, нестабильность далеко не всегда очевидна. Ни один заказчик или менеджер продукта на ранней стадии существования проекта не выдаст вам список требований к системе типа «Это может измениться, а это точно изменится позднее, зато это не изменится никогда». Окружающий мир (будь то заказчики, руководство или маркетинг) всегда представляет требования, выраженные в категориях функциональности: «Система должна делать то и это». Даже вам, читателю этой книги, скорее всего, будет непросто усвоить эту концепцию, когда вы попытаетесь выявить области нестабильности в вашей текущей системе. Соответственно, декомпозиция на основе нестабильности занимает больше времени по сравнению с функциональной декомпозицией.

Следует учесть, что декомпозиция на основе нестабильности не означает, что на требования можно не обращать внимания. Чтобы выявить области нестабильности, необходимо проанализировать требования. Можно даже утверждать, что главной целью анализа требований является выявление областей нестабильности, и этот анализ потребует немалых усилий. И на самом деле это хорошая новость, потому что вам предоставляется возможность выполнить первый закон термодинамики. К сожалению, просто потрудиться над задачей недостаточно. Первый закон термодинамики не утверждает, что если вы над чем-то попотели, то будет создана ценность. С созданием ценности дело обстоит намного сложнее. Эта книга предоставляет в ваше распоряжение мощные интеллектуальные средства проектирования и анализа, включая структуру, рекомендации и проверенную методологию. С этими средствами у вас появляется шанс на вашем пути к созданию ценности. Тем не менее все равно придется тренироваться и бороться.

Проблема 2%

В любой наукоемкой области вам потребуется время, чтобы стать грамотным и знающим специалистом, и еще больше времени для достижения вершин. Этот принцип действует в самых разнообразных областях, от ремонта канализации до терапевтической медицины и программной архитектуры. В жизни вы часто отказываетесь от изучения некоторых областей знаний, потому что время и затраты на их освоение слишком сильно превосходят время и затраты, необходимые для привлечения эксперта. Например, при отсутствии хронических заболеваний человек работоспособного возраста болеет в среднем одну неделю в год. Неделя бездействия из-за болезни составляет приблизительно 2% от производственного года. Итак, что вы делаете, когда заболевае-

те, — открываете учебники по медицине и начинаете читать или обращаетесь к доктору? Кроме 2% времени, потери достаточно малы (а планка освоения специальности достаточно высока), так что любые варианты, кроме обращения к доктору, не имеют смысла. Затраты времени и усилий на то, чтобы самому стать доктором, не оправдаются.

Однако если вы болеете 80% рабочего времени, ситуация меняется. Возможно, вы будете тратить значительную часть своего времени на сбор информации о своем состоянии, возможных осложнениях и возможном лечении — нередко до такого состояния, когда вы начинаете спорить с докторами. Ваша природная склонность к анатомии и медицине осталась на том же уровне; изменилась только величина вложенных ресурсов (будем надеяться, что у вас никогда не будет повода осваивать медицину).

То же самое происходит тогда, когда ваша кухонная раковина засоряется где-то за измельчителем отходов и посудомоечной машиной. Как вы поступаете: идете в хозяйственный магазин, покупаете одно- и двухоборотные сифоны, различные переходники, несколько видов ключей, уплотнительные кольца и другие инструменты или просто вызываете сантехника? Здесь снова возникает проблема 2%: если раковина засорена менее 2% времени, то учиться решать эту проблему просто нерационально. Мораль: если вы тратите на любую сложную задачу менее 2% своего времени, то никогда не будете специалистом в ней.

В области архитектуры программных систем архитекторы занимаются декомпозицией полной системы на модули только в переломных точках цикла. Такие события случаются в среднем раз в несколько лет. Все остальные решения в промежутке между очередными «нулевыми точками» в лучшем случае имеют инкрементную природу, а в худшем — наносят ущерб для существующих систем. Сколько времени выделит руководитель архитектору для работы над архитектурой следующего проекта? Неделю? Две недели? Три недели?? Шесть недель??? Точный ответ не столь важен. С одной стороны, вы имеете циклы, продолжительность которых измеряется годами, с другой — операции, продолжительность которых измеряется неделями. Отношение продолжительности недели к году равно приблизительно 1 : 50, то есть те же 2%. Архитекторы на собственном горьком опыте узнали, что они должны оттачивать свое мастерство, чтобы подготовиться к тому 2-процентному окну. Теперь представьте начальника архитектора. Если архитектор проводит 2% времени за проработкой архитектуры системы, то какой процент своего времени проводит руководитель архитектора за управлением этим архитектором? Вероятно, ничтожную часть. А следовательно, руководитель никогда не научится качественно управлять работой архитектора в этой критической фазе. Он так и будет постоянно восклицать: «Не понимаю, почему это занимает столько времени! Почему нельзя просто сделать A, B и C?»

Вероятно, с выделением времени на правильное проведение декомпозиции будет столько же проблем, как и с самой декомпозицией, если не больше. Тем не

менее сложность этой задачи не отменяет необходимости ее решения. Собственно, именно из-за своей сложности она непременно должна быть решена. Позднее в этой книге будут описаны некоторые приемы для получения времени.

Эффект Даннинга–Крюгера

В 1999 году Дэвид Даннинг (David Dunning) и Джастин Крюгер (Justin Kruger) опубликовали свое исследование[1], которое убедительно продемонстрировало, что люди, несведущие в некоторой области, склонны смотреть на нее сверху вниз и считать ее менее сложной, рискованной или требовательной, чем на самом деле. Это когнитивное искажение не имеет отношения к интеллекту или опыту в других областях. Если вы в чем-то плохо разбираетесь, то вы никогда не считаете эту область более сложной, чем на самом деле, — наоборот!

Когда руководитель воздевает руки к небу со словами: «Не понимаю, почему это занимает столько времени», он *действительно* не понимает, почему вы не можете сделать сначала A, потом B и потом C. Вас не должно это раздражать. Вы должны предвидеть такое поведение и справляться с ним, просвещая руководство и коллег, которые (по их собственному признанию) ничего не понимают.

Борьба с безумием

Альберту Эйнштейну приписывают утверждение о том, что безумие — это делать одно и то же снова и снова, каждый раз ожидая иного результата. Поскольку руководитель обычно ожидает, что вы справитесь со своим делом лучше, чем в последний раз, вы должны указать на безумие постоянного стремления к функциональной декомпозиции и разъяснить преимущества декомпозиции на основе нестабильности. В конечном итоге, даже если вам не удастся убедить одного человека, вы не должны просто выполнять приказы и закапывать проект в преждевременную могилу. Все равно проводите декомпозицию на основе нестабильности. Ставкой является ваша профессиональная добросовестность (и в конечном итоге ваше душевное равновесие и здравый смысл в долгосрочной перспективе).

Выявление нестабильности

В оставшейся части этой главы представлены некоторые инструменты, которые должны применяться при поиске и выявлении нестабильности. Они

[1] Justin Kruger and David Dunning, «Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments», Journal of Personality and Social Psychology 77, no. 6 (1999): 1121–1134.

полезны и эффективны сами по себе, но не все они четко формализованы. В следующей главе представлена структура и ограничения, обеспечивающие возможность быстрого и воспроизводимого выявления областей нестабильности. Впрочем, в этом обсуждении всего лишь уточняются и прорабатываются идеи из этого раздела.

Нестабильность и изменчивость

Ключевой вопрос, который вызывает проблемы у многих новичков, — чем отличаются изменяющиеся аспекты системы от нестабильных? Не все, что может изменяться, также является нестабильным. Вы прибегаете к инкапсуляции нестабильности на уровне проектирования системы только тогда, когда она не связана ограничениями и без инкапсуляции в компоненте архитектуры ее сдерживание обойдется слишком дорого. С другой стороны, изменчивость описывает те аспекты, которые вы можете легко обработать в своем коде с использованием условной логики. При поиске нестабильности необходимо обращать внимание на те изменения и риски, которые будут создавать каскадный эффект в системе. Изменения не должны приводить к нарушениям архитектуры.

Оси нестабильности

Поиск областей нестабильности — процесс исследования, который происходит во время анализа требований и собеседований с ключевыми участниками проекта.

Существует простой прием, который я называю *осями нестабильности*. В нем анализируются способы использования системы клиентами. В этом контексте «клиентом» называется любой потребитель системы, будь то отдельный пользователь или целая коммерческая организация.

В любой отрасли ваша система может справляться с изменениями в двух направлениях. Первая ось — фиксированный клиент с течением времени. Даже если в настоящее время система идеально сочетается с потребностями текущего клиента, со временем бизнес-контекст этого клиента может измениться. Даже само использование системы клиентом будет часто приводить к изменениям требований, с учетом которых она была изначально написана[1]. Со временем также могут изменяться требования клиента и его ожидания от системы.

[1] Тенденция к изменению требований под воздействием решения, которое с этими требованиями изначально разрабатывалось, была впервые отмечена английским экономистом XIX века Уильямом Дживонсом (William Jevons) применительно к добыче угля. С тех пор она называется «парадоксом Дживонса». Другими проявлениями можно считать повышение потребления бумаги в цифровых офисах и ухудшение ситуации с пробками после повышения пропускной способности дороги.

Второе направление изменений — фиксированное время с разными клиентами. Представьте, что вы можете остановить время и проанализировать клиентскую базу; все ли клиенты используют систему в точности одинаково? Что одни из них делают не так, как другие? Нужно ли учитывать такие различия? Все изменения такого рода определяют вторую ось нестабильности.

При поиске потенциальной нестабильности в ходе собеседований очень полезно формулировать вопросы в категориях осей нестабильности (один клиент с течением времени, все клиенты в одно время). Такая формулировка вопросов поможет вам выявить области нестабильности. Если какой-то аспект не укладывается на оси нестабильности, его вообще не следует инкапсулировать, и в системе не должно быть структурного элемента, который бы ему соответствовал. Вероятно, создание такого блока указывает на применение функциональной декомпозиции.

Разбиение в ходе проектирования

Часто поиск областей нестабильности с использованием осей нестабильности становится итеративным процессом, чередующимся с разбиением системы в ходе самого проектирования. Для примера возьмем итерации проектирования на рис. 2.10.

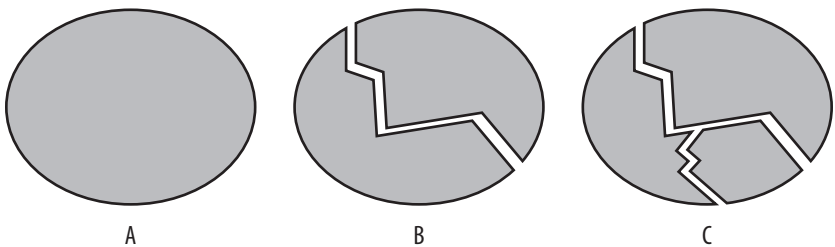


Рис. 2.10. Итерации проектирования по осям нестабильности

Возможно, первая версия предлагаемой архитектуры будет выглядеть так, как показано на диаграмме A, — один конгломерат, один большой компонент. Спросите себя: можно ли использовать один компонент в неизменном виде до бесконечности? Если ответ будет отрицательным, то почему? Часто это объясняется тем, что вы знаете, что со временем клиент захочет изменить некий конкретный аспект. В этом случае необходимо инкапсулировать этот конкретный аспект, что дает диаграмму B. Теперь спросите себя: можно ли использовать диаграмму B для всех клиентов? Если ответ будет отрицательным, определите, что именно клиенты хотят делать по-разному, и инкапсулируйте это; получается диаграмма C. Процесс разбиения продолжается до тех пор, пока не будут инкапсулированы все возможные точки на осях нестабильности.

Независимость осей

Как это обычно бывает, оси должны быть независимыми. То, что изменяется для одного клиента со временем, не должно изменяться в такой же степени по всем клиентам в одну точку времени, и наоборот. Если области изменений не удается изолировать по одной из осей, это часто свидетельствует о замаскированной функциональной декомпозиции.

Пример: декомпозиция дома на основе нестабильности

Оси нестабильности могут использоваться для инкапсуляции нестабильности дома. Для начала представьте дом и подумайте, как он изменяется во времени. Например, мебель: вы можете переставлять мебель в гостиной, время от времени добавлять новые или заменять старые предметы. Отсюда можно сделать вывод, что мебель нестабильна. Теперь возьмем бытовую технику. Возможно, со временем вы перейдете на устройства с низким энергопотреблением — скорее всего, вы уже заменили старый кинескопный телевизор плоской плазменной панелью или большим тонким OLED-телевизором. Это убедительный признак того, что в вашем доме бытовая техника нестабильна. Как насчет обитателей дома? Является ли этот аспект статическим? Приходят ли к вам гости? Может ли дом остаться пустым? Да, состав обитателей дома нестабилен. А что вы скажете о внешнем виде? Дом можно перекрасить, сменить занавески или провести благоустройство территории. Скорее всего, ваш дом будет подключен к некоторой инфраструктуре, от интернета до электропитания и средств безопасности. Ранее я уже упоминал о нестабильности питания в доме, но как насчет интернета? Когда-то мы использовали коммутируемый доступ для работы в интернете, потом перешли на DSL-кабели, а теперь пользуемся оптоволоконными линиями или спутниковыми подключениями. Хотя все эти варианты кардинально отличаются друг от друга, изменять способ отправки электронной почты в зависимости от типа подключения не хотелось бы. Нестабильность всех коммунальных услуг должна быть инкапсулирована. На рис. 2.11 показана возможная декомпозиция по первой оси нестабильности (постоянный клиент с изменением времени).

Рис. 2.11. Один дом с течением времени

А теперь можно ли сказать, что даже в один момент времени ваш дом в точности похож на каждый другой дом? Разные дома имеют разную структуру,

поэтому структура дома нестабильна. Даже если вы скопируете свой дом в другой город, то будет ли это тот же дом[1]?

Очевидно, ответ будет отрицательным. У дома будут другие соседи, на него будут распространяться другие городские строительные нормы, строительные кодексы и правила начисления налогов. На рис. 2.12 изображена возможная декомпозиция по второй оси нестабильности (разные клиенты в один момент времени).

Рис. 2.12. В фиксированный момент времени для разных домов

Обратите внимание на независимость осей. Город, в котором вы живете, изменяет свои правовые нормы, но изменения происходят достаточно медленно. Аналогичным образом вероятность появления новых соседей достаточно низка, пока вы живете в одном доме, но становится практически стопроцентной, если вы сравните свой дом с другим домом в то же время. Таким образом, связывание нестабильности с одной из осей является не явлением абсолютно исключенным, а скорее диспропорцией вероятности.

Также следует заметить, что компонент Нестабильность соседей может справиться с нестабильностью соседей в доме с течением времени так же легко, как и для разных домов в один момент времени. Связывание компонента с осью помогает изначально выявить нестабильность; для разных домов в один момент времени эта нестабильность всего лишь становится более очевидной.

Наконец, в резком контрасте с декомпозициями на рис. 2.6 и 2.7, в декомпозициях на рис. 2.11 и 2.12 нет компонента для приготовления еды или кухни. В декомпозиции на основе нестабильности требуемое поведение обеспечивается взаимодействием между разными инкапсулированными областями нестабильности. Приготовление обеда может быть результатом взаимодействия между обитателями дома, бытовой техникой, структурой и коммунальной инфраструктурой. Оси нестабильности — превосходная отправная точка, но это не единственный инструмент, который должен привлекаться для решения проблемы.

Решения, замаскированные под требования

Вернемся к функциональному требованию о поддержке функции приготовления еды в доме. Такие требования достаточно часто встречаются в специфика-

[1] Древние греки сформулировали этот вопрос в парадоксе Тесея (https://ru.wikipedia.org/wiki/Корабль_Тесея).

циях требований, и многие разработчики просто отображают его на компонент *Приготовление еды* в своей архитектуре. Однако приготовление еды само по себе не является требованием (хотя и входит в спецификацию требований). Оно всего лишь является возможным решением для требования о том, чтобы обитатели дома были накормлены. Чтобы удовлетворить это требование, также можно заказать пиццу на дом или сходить с семьей в ресторан.

Предъявление заказчиками решений, замаскированных под требования, — чрезвычайно частое явление. С функциональной декомпозицией после развертывания системы с компонентом *Приготовление еды* заказчик потребует добавить возможность заказа пиццы, что приведет либо к появлению нового компонента в системе, либо к разбуханию другого компонента. Затем вскоре последует требование «сходить в ресторан»; так возникает бесконечный цикл функций, вращающихся вокруг настоящего требования. При декомпозиции на основе нестабильности в ходе анализа требований следует выявить нестабильность в функциональности кормления обитателей дома и учесть ее при проектировании. Нестабильность кормления инкапсулируется в компоненте *Питание*, и с изменением вариантов питания ваше проектировочное решение не изменится.

Тем не менее хотя питание — более правильное требование, чем приготовление еды, оно также оказывается решением, замаскированным под требование. Что, если по правилам диеты обитатели дома должны лечь спать голодными? Требования о питании и диете могут быть взаимоисключающими. Вы можете выполнить только одно из них, но не оба сразу. Взаимоисключающие требования также встречаются очень часто.

Настоящее требование для любого дома — обеспечение благополучия своих обитателей, а не потребление ими калорий. В доме не должно быть слишком холодно или слишком тепло, слишком влажно или слишком сухо. Хотя заказчики могут обсуждать приготовление еды и ни разу не упомянуть о соблюдении температурного режима, вы должны распознать реальную нестабильность (благополучие) и инкапсулировать ее в компоненте *Благополучие* вашей архитектуры.

Так как многие спецификации требований переполнены решениями, замаскированными под требования, функциональная декомпозиция доводит ваши проблемы до максимума. Вы всегда будете гоняться за вечно изменяющимися решениями, так и не осознав истинные требования, на которых они базируются.

Тот факт, что в этих спецификациях требований содержатся решения, замаскированные под требования, в действительности оказывается благом, потому что пример приготовления еды в доме можно обобщить в полноценный метод анализа для выявления областей нестабильности. Для начала укажите на решения, замаскированные под требования, и спросите себя, существуют ли другие возможные решения? Если существуют, то какими были реальные требования и заложенная в них нестабильность? После того как вы выявите нестабиль-

ность, необходимо определить, является ли необходимость разрешения этой нестабильности истинным требованием или же остается решением, замаскированным под требование. После того как вы «отмоете» все решения, то, что у вас останется, может стать отличными кандидатами для декомпозиции на основе нестабильности.

Список нестабильностей

Прежде чем браться за декомпозицию и создавать архитектуру, следует просто откомпилировать список потенциальных областей нестабильности; это является естественной частью сбора и анализа требований. К списку следует подходить непредвзято. Спросите себя, что может изменяться по осям нестабильности. Выявите решения, замаскированные под требования, и примените другие средства, описанные далее в этой главе. Список станет мощным инструментом для отслеживания ваших наблюдений, он также поможет вам привести в порядок мысли. Пока не следует определяться с фактическим проектировочным решением; вы всего лишь составляете список. Учтите, что хотя на проектирование системы не должно уходить более нескольких дней, выявление правильных областей нестабильности может занять существенно больше времени.

Пример: торговая система на основе нестабильности

В случае с приведенными ранее требованиями для системы торговли акциями следует начать с составления списка возможных областей нестабильности, а также обоснований по каждому пункту:

- *Нестабильность пользователей.* Брокеры обслуживают клиентов, работая с их портфелями ценных бумаг. Для клиента также с большой вероятностью представляет интерес текущее состояние его средств. Хотя он может позвонить брокеру или написать ему письмо, в более правильном варианте клиент входит в систему для того, чтобы просмотреть текущий баланс и данные о текущих операциях. Хотя в требованиях ничего не говорится о доступе клиента (требования предназначены для профессиональных брокеров), такую возможность следует предусмотреть. Вероятно, клиенты не смогут выполнять операции, но они должны иметь возможность просмотреть состояние своих счетов. Также в системе могут присутствовать системные администраторы. Таким образом, существует нестабильность по типу пользователя.
- *Нестабильность клиентских приложений.* Нестабильность пользователей часто проявляется в нестабильности типа клиентского приложения и технологии. Для внешних клиентов, проверяющих свой баланс, может быть достаточно простой веб-страницы. Тем не менее профессиональный брокер предпочтет многоэкранное полнофункциональное настольное приложение

с рыночными трендами, подробной информацией о счетах, лентой котировок, каналом новостей, проекциями электронных таблиц и собственной информацией фирмы. Возможно, другие пользователи захотят просматривать тренды на мобильных устройствах различных типов.

- *Нестабильность безопасности*. Нестабильность пользователей подразумевает нестабильность способа аутентификации пользователей в системе. Количество внутренних брокеров может быть небольшим, от нескольких десятков до нескольких сотен. Однако у системы могут быть миллионы конечных пользователей. Внутренние брокеры могут положиться на доменные учетные записи для аутентификации, но для миллионов клиентов, обращающихся к информации из интернета, такой вариант не подойдет. Возможно, для интернет-пользователей будет достаточно простого имени и пароля, а может быть, потребуется система сложной единой федеративной безопасности. Аналогичная нестабильность существует со способами авторизации. Короче говоря, безопасность нестабильна.

- *Нестабильность уведомлений*. В требованиях указано, что система должна отправлять сообщение электронной почты после каждого запроса. Но что, если сообщение будет отклонено? Должна ли система переключиться на резервный вариант с бумажным письмом? А может, вместо электронной почты следует отправить текстовое сообщение или факс? Требование об отправке электронной почты — решение, замаскированное под требование. Реальное требование заключается в уведомлении пользователей, но транспортная среда уведомлений нестабильна. Также существует нестабильность в отношении того, кто получает сообщение: один пользователь или несколько пользователей, получающих одно уведомление по любому транспортному каналу. Возможно, конечный пользователь предпочтет электронную почту, тогда как адвокат по вопросам налогообложения предпочтет документированное заявление на бумаге. Также существует нестабильность в отношении того, кто публикует уведомление.

- *Нестабильность хранения*. В требованиях оговорено использование локальной базы данных. Тем не менее со временем все больше систем мигрирует на облачные платформы. В биржевой торговле нет ничего такого, что бы не позволяло пользоваться преимуществами стоимости и экономии на масштабах облачных технологий. Требование об использовании локальной базы данных в действительности является еще одним решением, замаскированным под требование. Более правильным требованием будет *постоянное хранение данных*, которое подразумевает нестабильность в вариантах хранения. Тем не менее большинство пользователей составляют люди, которые ограничиваются выполнением запросов, доступных только для чтения. Отсюда следует, что система сильно выиграет от использования кэша в памяти. Более того, некоторые облачные технологии используют распределенную хеш-таблицу в памяти, которая обладает такой же гибкостью, как

и традиционный способ постоянного хранения на базе файлов. Требование постоянного хранения данных исключает два последних варианта, потому что постоянное хранение данных остается решением, замаскированным под требование. Реальное требование заключается лишь в том, что система не должна терять данные или что система должна каким-то образом хранить данные. Конкретная реализация этих требований — вопрос реализации со значительной степенью нестабильности, от локальной базы данных до удаленного кэша в памяти, хранимого в облаке.

- *Нестабильность режима подключения и синхронности*. В текущих требованиях указан жестко синхронный способ заполнения веб-формы и ее отправки. Отсюда следует, что брокеры смогут выполнять только один запрос в любой момент времени. Но чем больше операций выполняют брокеры, тем больше денег они заработают. Если запросы не зависят друг от друга, то почему бы не выполнять их асинхронно? Если запросы отложены по времени (операции, запланированные на будущее), почему бы не создать очередь обращений к системе для снижения нагрузки? При использовании асинхронных вызовов (включая очереди вызовов) запросы могут выполняться в произвольном порядке. Режим подключения и синхронность нестабильны.

- *Нестабильность продолжительности сеанса и устройств*. Некоторые пользователи завершают сделку за один короткий сеанс. Однако брокеры получают максимальный доход при выполнении сложных операций с распределением и хеджированием рисков, с разными видами акций и секторами, на внутренних или внешних рынках и т. д. Проведение таких сделок может занимать много времени, от нескольких часов до нескольких дней. Такие долгосрочные взаимодействия с большой вероятностью будут охватывать несколько сеансов работы с системой — возможно, с нескольких физических устройств. В продолжительности взаимодействий существует нестабильность, которая, в свою очередь, инициирует нестабильность устройств и задействованных подключений.

- *Нестабильность торговых позиций*. Как упоминалось ранее, со временем пользователи могут захотеть торговать не только акциями, но и товарами, облигациями, валютой и даже фьючерсными контрактами. Сами торговые позиции нестабильны.

- *Нестабильность процесса*. Если торговые *позиции нестабильны*, то нестабильными будут и действия при выполнении основных фаз операции. Покупка и продажа акций, планирование заказов и т. д. очень сильно отличаются от продажи товаров, облигаций или валют. Таким образом, процесс продажи нестабилен. Аналогичным образом нестабилен и процесс анализа операций.

- *Нестабильность локального контекста и законодательства*. Возможно, со временем система будет развернута в других регионах. Нестабильность локального контекста очень сильно влияет на правила торговли, локализацию пользовательского интерфейса, список торговых позиций, налогообложение и соблюдение нормативных требований. Следовательно, локальный контекст и действующие в нем нормы нестабильны.
- *Нестабильность канала данных рынка*. Источник рыночных данных может изменяться со временем. Разные каналы используют разный формат, стоимость, частоту обновления, коммуникационные протоколы и т. д. Они могут выдавать несколько отличающиеся цены одного вида акций в один момент времени. Каналы могут быть внешними (например, Bloomberg или Reuters) или внутренними (например, моделируемые рыночные данные для тестирования, диагностики или анализа торговых алгоритмов). Каналы данных нестабильны.

Ключевое наблюдение

Приведенный выше список ни в коем случае не содержит всего, что может измениться в системе биржевой торговли. Его цель — указать на те аспекты, которые могут измениться, и представить общий подход к поиску нестабильных аспектов. Некоторые области нестабильности могут выходить за рамки проекта. Они могут быть исключены экспертами в предметной области как слишком маловероятные или же могут быть слишком сильно связаны с природой бизнеса (например, переходом от акций к валютам или иностранным рынкам). Однако мой опыт показывает, что для проекта очень важно как можно ранее определить области нестабильности и отобразить их в декомпозиции. Выделение проекта в архитектуре не стоит практически ничего. Позднее вы должны решить, стоит или нет тратить усилия на его проектирование и конструирование. Тем не менее теперь вы по крайней мере знаете, как справиться с этой возможностью.

Декомпозиция системы

После того как области нестабильности будут определены, необходимо инкапсулировать их в компонентах архитектуры. Один из возможных вариантов декомпозиции изображен на рис. 2.13.

Переход от списка областей нестабильности к компонентам архитектуры практически никогда не бывает однозначным. Иногда один компонент может инкапсулировать более одной области. Некоторым областям нестабильности могут соответствовать не компоненты, а рабочие концепции, такие как очереди или публикация событий. В других случаях нестабильность области может инкапсулироваться в стороннем сервисе.

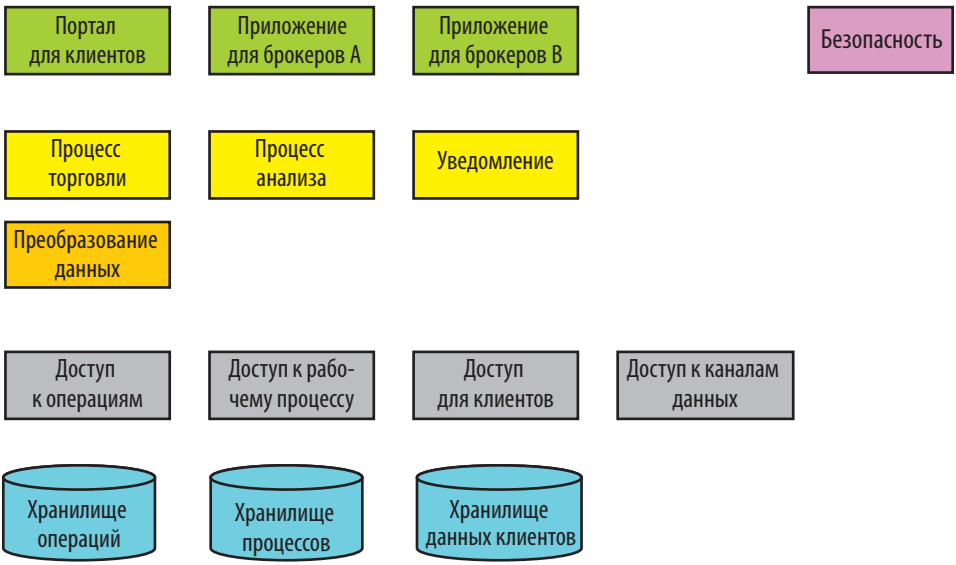


Рис. 2.13. Декомпозиция на основе нестабильности для торговой системы

При проектировании всегда начинайте с простых и легких решений. Такие решения устанавливают ограничения в системе и упрощают последующие решения. В приведенном примере некоторые соответствия устанавливаются легко. Нестабильность в хранении данных инкапсулируется за компонентами доступа к данным, которые не раскрывают информацию о том, где находится хранилище и какая технология используется для обращения к нему. На рис. 2.13 обратите внимание на ключевую абстракцию: использование термина «Хранилище» вместо «Базы данных». Если в качестве реализации (в соответствии с требованиями) выбрана локальная база данных, в архитектуре нет ничего, что бы исключало другие варианты: простую файловую систему, кэш или облачное хранение. Если технология хранения изменится, она будет инкапсулирована в соответствующем компоненте доступа (например, «Доступ к операциям») и не повлияет на другие компоненты, включая другие компоненты доступа к данным. Это позволит вам сменить способ хранения с минимальными последствиями.

Нестабильность уведомления клиентов инкапсулирована в компоненте *Уведомление*. Этот компонент знает, как уведомить каждого клиента и какие клиенты подписаны на то или иное событие. В простых сценариях можно достаточно эффективно управлять уведомлениями при помощи сервиса публикации/подписки (Pub/Sub) общего назначения вместо специального компонента *Уведомление*. Однако в этом случае, скорее всего, будут действовать некоторые бизнес-правила для типа транспорта и характера рассылки. Компонент Уведомление также может использовать некую разновидность сервиса публикации/

подписки в своей внутренней реализации, но это внутренняя подробность реализации, нестабильность которой также инкапсулируется в компоненте *Уведомление*.

Нестабильность процесса торговли инкапсулируется в компоненте *Процесс торговли*. Этот компонент инкапсулирует нестабильность продаваемых позиций (акции или валюта), конкретные шаги по покупке или продаже, необходимую адаптацию для местных рынков, подробности требуемых отчетов и т. д. Обратите внимание: даже если продаваемые позиции фиксированы (то есть не имеют нестабильности), процесс торговли акциями может изменяться, что оправдывает применение компонента *Процесс торговли* для инкапсуляции нестабильности. Такое проектирование также опирается на концепцию хранения рабочих процессов (для реализации которой должен использоваться некий сторонний инструмент). Компонент загружает подходящий экземпляр рабочего процесса для каждого сеанса, выполняет с ним необходимые действия и снова сохраняет в *Хранилище процессов*. Эта концепция помогает инкапсулировать несколько видов нестабильности. Во-первых, разные торговые позиции теперь могут иметь разные торговые процессы. Во-вторых, возможно использование разных процессов для разных локальных контекстов. В-третьих, это позволяет поддерживать продолжительные процессы, охватывающие по несколько устройств и сеансов. Систему не интересует, какой промежуток времени прошел между двумя вызовами — секунды или дни. В любом случае система загружает экземпляр рабочего процесса, чтобы обработать следующий шаг. В этом проектировочном решении непрерывные односеансовые операции обрабатываются точно так же, как долгосрочные распределенные. Симметрия и последовательность — хорошие качества в системной архитектуре. Также обратите внимание на то, что доступ к хранилищу процессов инкапсулируется точно так же, как и доступ к хранилищу операций.

Один паттерн может использоваться как для процесса торговли акциями, так и для процесса анализа. Специализированный компонент *Процесс анализа* инкапсулирует нестабильность процессов анализа, и он может использовать то же Хранилище процессов. Нестабильность обращений к каналам рыночных данных инкапсулируется в компоненте *Доступ к каналам данных*. Этот компонент инкапсулирует способ обращения к данным, а также то, является ли сам канал внешним или внутренним. Нестабильность в формате и даже в значениях различных рыночных данных от различных каналов инкапсулируется в компоненте *Преобразование данных*. Оба компонента ослабляют связи других компонентов с каналами, предоставляя унифицированный интерфейс и формат, не зависящие от источника данных.

Компонент *Безопасность* инкапсулирует нестабильность различных способов аутентификации и авторизации пользователей. Во внутренней реализации он может получить регистрационные данные из локального хранилища или запросить их у распределенного поставщика.

Клиентом системы может быть приложение для брокера (Приложение для брокеров A) или мобильное приложение (Приложение для брокеров B). Клиенты могут воспользоваться специальным веб-сайтом (Портал для клиентов). Каждое клиентское приложение также инкапсулирует подробности и оптимальный способ отображения информации на целевом устройстве.

> **ПРИМЕЧАНИЕ** В предыдущем обсуждении использовалось неформальное отображение областей нестабильности на архитектуру. В следующей главе будет приведено описание архитектуры и список рекомендаций, которые сделают этот процесс намного более детерминированным.

Как противостоять зову сирен

Обратите внимание: на рис. 2.13 отсутствует специализированный компонент построения отчетов. Для демонстрационных целей отчеты не были включены в список как область нестабильности (с точки зрения бизнеса). Следовательно, инкапсулировать в компоненте нечего. Добавление такого компонента стало бы проявлением функциональной декомпозиции. Тем не менее если вы никогда не применяли ничего, кроме функциональной декомпозиции, то скорее всего, вы услышите непреодолимый зов сирен, убеждающий вас добавить блок для отчетов. Даже если вы всегда создавали блок для отчетов или в вашей системе уже имеется готовый блок отчетов, — это не значит, что вам нужен блок отчетов.

В «Одиссее» Гомера — эпическом повествовании, написанном более 2500 лет назад, — Одиссей плывет домой мимо острова сирен. Сирены — прекрасные крылатые существа с ангельскими голосами. Никто не может устоять перед их песнями. Моряки прыгают в воду, сирены топят их в волнах и едят. Но прежде чем столкнуться со смертоносным соблазном, Одиссей (вы, архитектор) получает совет — заткнуть уши моряков (рядовые разработчики) воском и привязать их к веслам. Дело моряков — грести (писать код), чтобы у них не было даже возможности слушать зов сирен. Одиссей велел привязать себя к мачте корабля, чтобы не поддаться зову сирен (на рис. 2.14 изображен рисунок на вазе того исторического периода). Вы — Одиссей, а декомпозиция на основе нестабильности — ваша мачта. Сопротивляйтесь зову сирен — ваших старых вредных привычек.

Хотя области нестабильности необходимо инкапсулировать, не все, что может измениться, должно быть инкапсулировано. Иначе говоря, то, что может измениться, не всегда нестабильно. Классический пример — природа бизнеса; вы не должны пытаться инкапсулировать природу бизнеса. Почти во всех бизнес-областях приложение существует для удовлетворения некоторых потребностей бизнеса или его клиентов. Тем не менее природа бизнеса (а значит, и каждое приложение) обычно остается относительно постоянной. Компания, которая работала в некоторой отрасли в течение долгого времени, с большой вероятно-

стью останется в этой отрасли. Например, компания Federal Express работала и будет работать в области транспортировки и доставки грузов. Хотя теоретически Federal Express может переключиться на область здравоохранения, такое потенциальное изменение не является чем-то, что должно инкапсулироваться.

Рис. 2.14. Привязанный к мачте (изображение: Werner Forman Archive/Shutterstock)

В ходе декомпозиции системы необходимо выявить области нестабильности, которые должны или не должны инкапсулироваться (например, природа бизнеса). Иногда различить их может быть достаточно сложно. Существуют два простых признака, по которым можно определить, является ли то, что может измениться, частью природы бизнеса. Первый признак — редкость возможных изменений. Да, теоретически такое возможно, но вероятность такого события очень низка. Второй признак — любые попытки инкапсуляции изменений могут быть реализованы плохо. Никакие затраты времени или усилий не позволят инкапсулировать этот аспект так, чтобы вы могли бы гордиться таким решением.

Например, рассмотрим проектирование простого жилого дома на участке земли. Когда-нибудь в будущем владелец дома может захотеть расширить дом до 50-этажного небоскреба. Инкапсуляция возможных изменений в проектиро-

вочном решении дома породит результат, который очень сильно отличается от типичного проекта жилого дома. Вместо фундамента с неглубокой заливкой основание дома должно включать десятки опор, забитых на десятки метров и способных выдержать вес здания. Такой фундамент сможет выдержать как семейный жилой дом, так и небоскреб. Кроме того, распределительный щит должен быть способен выдержать тысячи ампер, и с большой вероятностью дому потребуется собственный трансформатор. Компания водоснабжения может подавать воду в дом, но вам придется выделить комнату под мощный насос, способный поднять воду на 50 этажей. Система канализации должна обслуживать обитателей на 50 этажах. Вам придется пойти на эти колоссальные затраты для простого дома, рассчитанного на одну семью.

Когда проектирование будет завершено, фундамент будет инкапсулировать изменение веса здания, распределительный щит будет инкапсулировать потребности как небольшого дома, так и 50-этажного здания, и т. д. Однако при этом будут нарушены оба признака. Во-первых, сколько домовладельцев в вашем доме перестраивали свои дома в небоскребы? Насколько распространено такое явление? В крупном городе с миллионами домов такое может происходить раз в несколько лет, вследствие чего изменения становятся очень редкими — одно на миллион. Во-вторых, располагаете ли вы средствами (выделенными изначально для одного дома) для правильного выполнения всех инкапсуляций? Стоимость одной опоры может превысить стоимость семейного дома. Любая попытка инкапсулировать будущую перестройку в небоскреб будет выполнена плохо, и она не будет ни практичной, ни эффективной по затратам.

Преобразование дома, рассчитанного на одну семью, в 50-этажное здание изменяет природу бизнеса. Здание уже не относится к области предоставления жилья для семей. Теперь оно относится к области использования в качестве отеля или офисного здания. Когда агент по земельной собственности приобретает участок земли для выполнения такой перестройки, он обычно планирует снести здание, вырыть старый фундамент и начать все заново. Изменение природы бизнеса позволяет уничтожить старую систему и начать с нуля. Важно заметить, что контекст природы бизнеса отчасти неоднозначен. Таким контекстом может быть бизнес целой компании, одного отдела или подразделения компании или даже конкретное приложение. Все эти аспекты не должны инкапсулироваться.

Умозрительное проектирование

Умозрительное проектирование является разновидностью попыток инкапсулировать природу бизнеса. Стоит вам принять принцип декомпозиции на основе нестабильности, как вы начнете видеть потенциальные нестабильности повсюду — и легко сможете переусердствовать. В самом крайнем случае вы пытаетесь инкапсулировать все и повсюду. Ваше решение будет состоять из

Рис. 2.15. Умозрительное проектирование (изображение: Gercen/Shutterstock)

множества структурных элементов — четкий признак некачественного проектирования. Для примера возьмем предмет на рис. 2.15.

На рисунке изображены женские туфли на высоком каблуке, готовые к подводному плаванию. Если леди в вечернем платье развлекает своих гостей на вечеринке, насколько вероятно, что она извинится, отойдет, наденет акваланг и займется подводным плаванием на рифе? Остаются ли эти туфли такими же элегантными, как традиционные туфли? Не уступают ли они по эффективности обычным ластам, когда вы плаваете или наступаете на острый коралл? Хотя использование туфель на рис. 2.15 возможно, оно крайне маловероятно. Кроме того, все, что оно пытается сделать, делается очень плохо из-за попытки инкапсулировать изменение природы обуви, от модного аксессуара до принадлежности для подводного плавания — никогда не пытайтесь делать что-то подобное. Такие попытки заводят вас в ловушку спекулятивного проектирования. Практически все решения такого рода представляют собой легкомысленные спекуляции относительно будущих изменений вашей системы (то есть изменения природы бизнеса).

Проектирование для конкурентов

Другой полезный способ выявления нестабильности — попытка спроектировать систему для вашего конкурента (или другого отдела вашей компании). Представьте, что вы работаете на должности ведущего архитектора системы следующего поколения для Federal Express. Ваш главный конкурент — компания UPS. И Federal Express, и UPS занимаются доставкой посылок. Обе компании получают оплату, планируют получение и доставку грузов, обеспечивают отслеживание пакетов и целостность грузов, управляют маршрутизацией посылок на грузовиках и самолетах. Задайте себе следующий вопрос: сможет ли Federal Express использовать программную систему, которой пользуется UPS? Сможет ли UPS использовать систему, которую хочет построить Federal Express? Если наиболее вероятный ответ — «нет», начните составлять список всех препятствий для повторного использования или возможностей

расширения системы. Хотя обе компании на абстрактном уровне предоставляют одинаковые услуги, модели ведения бизнеса у них отличаются. Например, Federal Express может планировать маршруты одним способом, а UPS — другим. В данном случае планирование перевозок, вероятно, становится нестабильным, потому что если что-то можно сделать двумя разными способами, то, скорее всего, таких способов найдется еще много. Планирование перевозки следует инкапсулировать и создать в архитектуре специальный компонент для этой цели. Если Federal Express когда-нибудь в будущем начнет планировать перевозки так же, как UPS, изменения теперь будут ограничены в одном компоненте; такие изменения будут реализовываться проще и с влиянием только на реализацию этого компонента, но не на декомпозицию в целом. Тем самым гарантируется, что ваша система выдержит проверку временем.

Обратная ситуация также истинна. Если вы и ваш конкурент (или еще лучше — все конкуренты) выполняете некоторую операцию или серию операций одинаково и нет ни малейшей вероятности того, что ваша система может выполнять ее как-то иначе, выделять для этой операции отдельный компонент в архитектуре не следует, поскольку это привело бы к созданию функциональной декомпозиции. Когда вы встречаете нечто такое, что ваши конкуренты делают одинаково, скорее всего, речь идет о природе бизнеса, которая, как объяснялось ранее, инкапсулироваться не должна.

Нестабильность и долговечность

Нестабильность тесно связана с долговечностью. Чем дольше компания (или приложение) делает что-то определенным образом, тем больше вероятность того, что в будущем она будет это делать точно так же. Иными словами, долгосрочное обычно не меняется и продолжает существовать до тех пор, пока все-таки не будет изменено или заменено чем-то другим. Вы должны разработать решение, которое допускает такие изменения, даже если на первый взгляд они не зависят от текущих требований.

Вы даже можете предположительно оценить, через какое время может возникнуть такое изменение, при помощи простой эвристики: способность организации (или клиента, или рынка) инициировать или поглощать изменения остается более или менее постоянной, потому что она связана с природой бизнеса. Например, IT-отдел больницы более консервативен и хуже воспринимает изменения, чем зарождающийся стартап. Таким образом, чем чаще что-то изменяется, тем выше вероятность будущих изменений — но с приблизительно такой же частотой. Например, если каждые 2 года компания меняет свою систему расчета зарплат, то с большой вероятностью компания изменит ее в течение следующих 2 лет. Если проектируемая вами система должна взаимодействовать с системой расчета зарплат, а горизонт использования вашей системы превышает 2 года, вы должны инкапсулировать нестабильность системы расчета зарплат и запланировать ограничение возможных изменений. Эффект

изменения зарплатной системы необходимо учитывать, даже несмотря на то что это изменение никогда не предоставлялось вам среди явных требований. Вы должны стремиться к инкапсуляции изменений, происходящих на протяжении срока жизни системы. Если предполагаемая продолжительность жизни составляет от 5 до 7 лет, хорошей отправной точкой станет выявление всех аспектов, изменившихся в предметной области приложения за последние 7 лет. Можно ожидать, что на протяжении такого же периода в системе произойдут аналогичные изменения.

Непременно проанализируйте долговечность всех систем и подсистем, с которыми должно взаимодействовать ваше решение. Например, если система планирования корпоративных ресурсов (ERP) изменяется каждые 10 лет, последнее изменение ERP состоялось 8 лет назад, а горизонт новой системы составляет 5 лет, то можно с большой вероятностью ожидать, что ERP изменится на протяжении срока жизни вашей системы.

О важности практики

Если вы проводите за какой-то деятельностью только 2% времени, вы никогда не будете хорошо разбираться в ней, независимо от врожденного интеллекта или использованной методологии. Чтобы предположить, что раз в несколько лет кто-то может подойти к доске, нарисовать несколько линий и понять суть архитектуры, необходима просто невероятная самоуверенность. Все профессионалы, будь то доктор, пилот, сварщик или адвокат, ожидают, что они могут освоить свое ремесло упорной тренировкой. Вам бы не хотелось оказаться пассажиром на борту самолета, пилот которого отработал всего десяток летных часов, как и быть первым пациентом у врача. Пилоты коммерческих авиалиний проводят (многие) часы на имитаторах и проходят обучение в сотнях рейсов под руководством пилотов-ветеранов. Врачи проводят вскрытие бесчисленных трупов до того, как смогут прикоснуться к первому пациенту, и даже тогда они находятся под тщательным контролем.

Выявление областей нестабильности является приобретаемым навыком. Вряд ли хоть один архитектор программных систем прошел изначальную подготовку в декомпозиции на основе нестабильности, а в абсолютном большинстве систем и проектов применяется функциональная декомпозиция (с ужасающими результатами). Лучший способ освоения декомпозиции на основе нестабильности — практика. Собственно, только так можно решить проблему 2%. Несколько возможных вариантов:

- Практикуйтесь на хорошо известных вам программных системах — например, типичной страховой компании, мобильном банке или интернет-магазине.
- Проанализируйте свои прошлые проекты. По прошествии времени вы уже знаете их основные болевые точки. Была ли архитектура этих прошлых

проектов реализована функционально? Что изменилось? Какие каскадные эффекты вызывали эти изменения? Если бы эта нестабильность была инкапсулирована, позволило бы это лучше справиться с изменениями?

- Присмотритесь к своему текущему проекту. Возможно, его еще не поздно спасти: был ли он спроектирован функционально? Сможете ли вы перечислить области нестабильности и предложить улучшенную архитектуру?
- Проанализируйте непрограммные системы: велосипед, ноутбук, дом. Выявите в них области нестабильности.

Затем проделайте все то же, потом еще раз. Практикуйтесь раз за разом. После того, как будут проанализированы от трех до пяти систем, вы получите общее представление о методе. К сожалению, умение выявлять области нестабильности — не тот навык, который можно приобрести наблюдением за другими. Нельзя научиться кататься на велосипеде по книге; вы должны несколько раз сесть на велосипед (и упасть с него). Это справедливо и для декомпозиции на основе нестабильности. Но лучше потерпеть неудачу в ходе обучения, чем экспериментировать на живых пациентах.

3 Структура

В предыдущей главе обсуждался универсальный принцип проектирования — декомпозиция на основе нестабильности. Этот принцип управляет проектированием всех практических систем — от домов до ноутбуков, от воздушных лайнеров до вашего собственного тела. Чтобы выжить и добиться успеха, они инкапсулируют нестабильность своих компонентов. Архитекторы программных систем занимаются проектированием только программных систем. К счастью, во всех этих системах существуют общие области нестабильности. За прошедшие годы я обнаруживал эти общие области нестабильности в сотнях систем. Более того, существуют типичные взаимодействия, ограничения и отношения, связывающие эти общие области нестабильности во время выполнения. Если вы научитесь распознавать их, это позволит вам строить правильные системы быстро и эффективно.

На основании этого наблюдения Метод предоставляет шаблон для областей нестабильности, рекомендации для взаимодействий, также предлагает рабочие паттерны. При этом Метод выходит за рамки простой декомпозиции. Возможность предоставления таких обобщенных рекомендаций и структуры для большинства программных систем может показаться неправдоподобной. Действительно ли такие обобщения могут применяться в разнообразных программных системах? Дело в том, что хорошие архитектуры допускают применение в разных контекстах. Например, мышь и слон совершенно не похожи, но они имеют идентичную архитектуру. Однако детализированное строение слона и мыши не имеет ничего общего. Точно так же Метод предоставляет в ваше распоряжение системную архитектуру, но не подробное проектировочное решение.

Эта глава посвящена способу структурирования систем по правилам Метода, его преимуществам и влиянию на архитектуру. В ней будет приведена классификация сервисов по их семантике и сопутствующие рекомендации, а также способы определения иерархической структуры. Кроме того, четкая, последовательная номенклатура для компонентов архитектуры и связей между ними открывает два других преимущества. Во-первых, она становится хорошей отправной точкой. Вам все равно придется потрудиться над ней, но по крайней

мере понятно, с чего начинать. Во-вторых, она повышает качество коммуникаций, потому что вы должны сообщить свои намерения при проектировании другим архитекторам или разработчикам. Даже общение с самим собой очень ценно, поскольку оно поможет прояснить ваши собственные мысли.

Сценарии использования и требования

Прежде чем углубляться в архитектуру, рассмотрим требования. В большинстве проектов, если они вообще берут на себя труд по фиксации требований, используются функциональные требования. Функциональные требования просто заявляют необходимую функциональность типа «Система должна делать А». На самом деле это плохой способ определения требований, потому что системная реализация функциональности А остается открытой для интерпретации. На самом деле функциональные требования открывают несколько возможностей для ошибочных интерпретаций между клиентами и маркетингом, между маркетингом и технической службой и даже между разработчиками. Такие неоднозначности обычно остаются до того момента, когда будут потрачены значительные усилия на разработку и развертывания системы, то есть до момента, когда их исправление обходится дороже всего.

Требования должны отражать требуемое поведение вместо требуемой функциональности. Вы должны определить, как система должна работать, а не то, что она должна делать, в чем, по мнению некоторых, проявляется сущность сбора требований. Как и во многих других случаях, это потребует дополнительной работы и усилий (того, чего люди обычно стараются избегать), так что сбор требований в таком виде станет нелегким делом.

Требуемое поведение

Сценарий использования является выражением требуемого поведения, то есть того, как система должна подойти к выполнению некоторой работы и созданию ценности для бизнеса. Как следствие, сценарий использования представляет собой конкретную последовательность операций в системе. Сценарии использования обычно бывают длинными и имеют описательную природу. Они могут описывать взаимодействия конечного пользователя с системой, или взаимодействие системы с другими системами, или выполнение служебных операций. Этот аспект важен, потому что в любой хорошо спроектированной системе, даже при умеренном размере и сложности, пользователи взаимодействуют и видят лишь малую часть системы, то есть только верхушку айсберга. Основная часть системы остается под водой, и сценарии использования следует продумать и для нее.

Сценарии использования могут сохраняться в текстовом или в графическом виде. Текстовые сценарии использования легко создаются, что, безусловно, является

преимуществом. К сожалению, использование текста для описания сценариев использования — не лучший способ, потому что сценарии могут быть слишком сложными для точного отражения в тексте. Но настоящая проблема с текстовым сценарием заключается в том, что мало кто утруждается чтением даже простого текста, и на то есть веская причина. Чтение является искусственной деятельностью для человеческого мозга, потому что мозг не приспособлен для простого усвоения и обработки сложных идей через текст. Человечество читает всего 5000 лет — этого недостаточно, чтобы мозг успел наверстать эволюционный пробел (впрочем, мы благодарны вам за усилия по чтению этой книги).

Сценарии использования лучше описывать в графическом виде на диаграмме (рис. 3.1). Человек обрабатывает изображения с невероятной быстротой, потому что почти половина человеческого мозга представляет собой огромный процессор для обработки видеоинформации. Вы можете воспользоваться этим процессором, чтобы донести идеи для вашей аудитории.

С другой стороны, построение графического представления сценариев использования может быть весьма трудоемким, особенно в большом количестве. Многие сценарии использования могут быть достаточно простыми для понимания без диаграмм. Например, диаграмма на рис. 3.1 может быть с таким же успехом представлена в текстовом виде. Мое эмпирическое правило: присутствие вложенной конструкции «если» указывает на то, что сценарий использования лучше нарисовать. Ни один читатель не сможет мысленно разобрать предложение с вложенными «если». Вместо этого он либо неоднократно перечитает сценарий использования, либо, что более вероятно, возьмет бумагу и ручку и попытается построить визуальное представление сценария использования самостоятельно. При этом читатели будут интерпретировать поведение, что также открывает возможность неправильной интерпретации. Когда читатель делает пометки на полях вашего текстового сценария, значит, вам изначально нужно было предоставить визуализацию. С диаграммами читателю также будет проще отслеживать большое количество вложенных «если» в сложных сценариях использования.

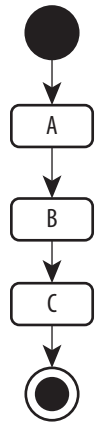


Рис. 3.1. Диаграмма сценария использования

Диаграммы активности

Метод рекомендует использовать диаграммы активности (или диаграммы деятельности)[1] для графического представления сценариев использования, прежде всего из-за того, что диаграммы активности способны отразить аспек-

[1] https://ru.wikipedia.org/wiki/Диаграмма_деятельности

ты поведения, критичные по времени, на что не способны блок-схемы и другие виды диаграмм. На блок-схеме невозможно представить параллельное выполнение, блокирование или ожидание наступления некоторого события. С другой стороны, в диаграммах активности имеется встроенная поддержка параллелизма.

Например, на рис. 3.2 представлена интуитивно понятная обработка параллельного выполнения как реакции на событие, причем вам даже не нужно просматривать список условных обозначений. Обратите внимание, насколько легко отслеживается вложенное условие.

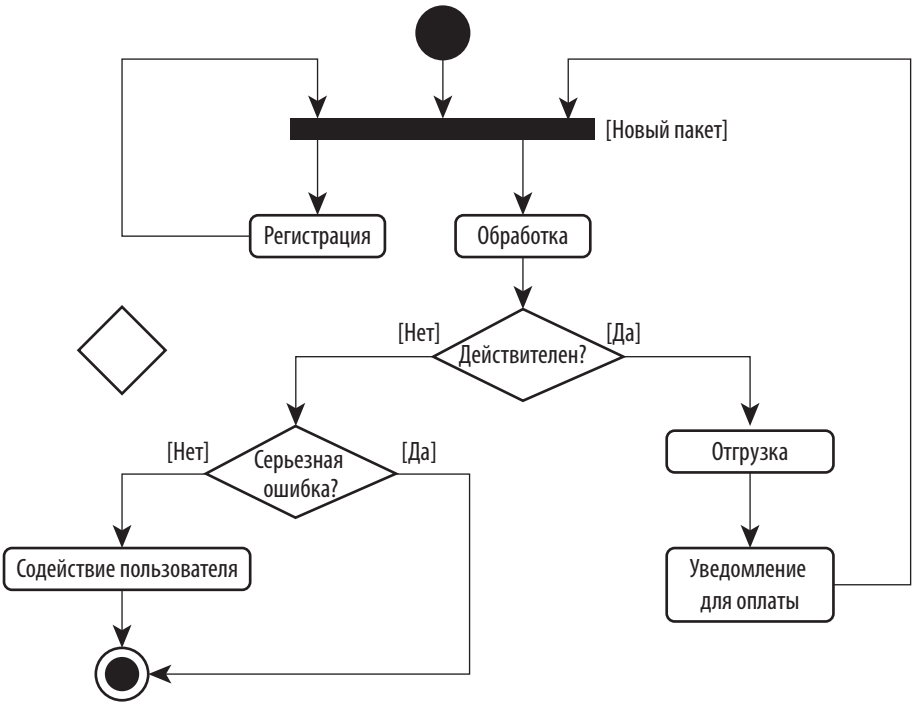


Рис. 3.2. Диаграмма активности

ВНИМАНИЕ Не путайте диаграммы активности с диаграммами сценариев использования.[1] Диаграммы сценариев использования ориентированы на пользователя, и их правильнее было бы назвать «диаграммами пользовательских сценариев». Диаграммы сценариев использования также не поддерживают концепции времени или последовательности.

[1] https://ru.wikipedia.org/wiki/Сценарий_использования

Многоуровневый подход

Программные системы обычно проектируются по уровням, а работа Метода сильно зависит от уровней. Уровни позволяют создавать уровни инкапсуляции. Каждый уровень инкапсулирует свои собственные нестабильности от уровней, находящихся выше и ниже в иерархии. Сервисы внутри уровней инкапсулируют нестабильность друг от друга, как показано на рис. 3.3.

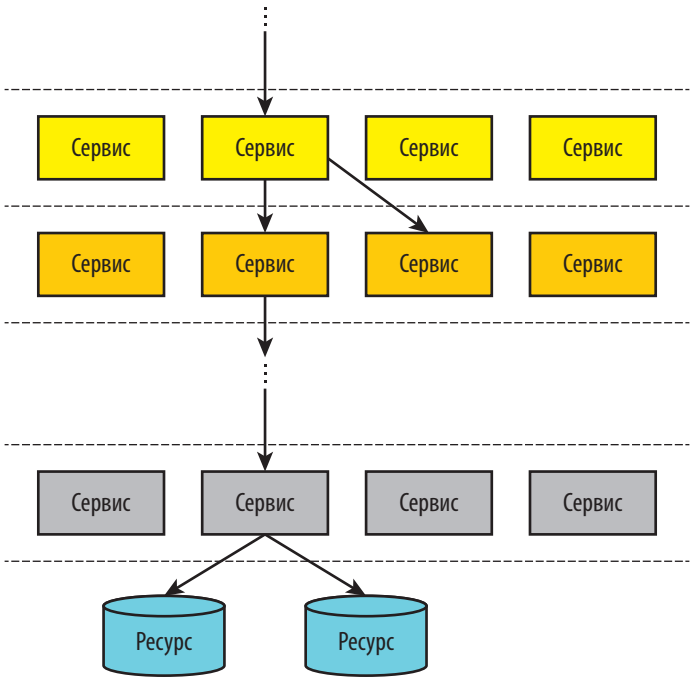


Рис. 3.3. Сервисы и уровни

Даже простые системы должны проектироваться по уровням, чтобы вы могли пользоваться преимуществами инкапсуляции. Теоретически чем больше уровней, тем лучше инкапсуляция. В практических системах обычно присутствует небольшой набор уровней, который завершается уровнем физических ресурсов (таких, как хранилище данных или очередь сообщений).

Использование сервисов

Рекомендуемый способ взаимодействия между уровнями — вызов сервисов. Конечно, преимуществами структуры Метода и декомпозиции на основе нестабильности можно пользоваться даже с обычными классами, но сервисы предоставляют особые преимущества. Вопрос о том, какую именно техноло-

гию и платформу использовать для реализации сервисов, вторичен. Использование сервисов (при условии, что выбранная технология это позволяет) немедленно открывает следующие преимущества:

- *Масштабируемость*. Экземпляры сервисов могут создаваться множеством разных способов, в том числе и на уровне вызовов. Это позволяет поддерживать очень большое количество клиентов без создания пропорциональной нагрузки на внутренние ресурсы, так как количество экземпляров сервисов будет равно количеству обрабатываемых вызовов.
- *Безопасность*. Все сервисно-ориентированные платформы относятся к безопасности как к первоочередному аспекту. По этой причине они проводят аутентификацию и авторизацию всех вызовов — не только вызовов сервисов из клиентского приложения, но и вызовов между сервисами. Вы даже можете использовать механизм распространения идентификационных данных для поддержки паттерна «цепочка доверия».
- *Пропускная способность и доступность*. Сервисы могут принимать вызовы в очередях, что позволяет обрабатывать очень большие объемы сообщений простой постановкой в очередь избыточной нагрузки. Очереди вызовов также улучшают доступность, поскольку одна входная очередь может обрабатываться несколькими экземплярами сервисов.
- *Скорость реакции*. Сервисы могут помещать вызовы в буфер, чтобы избежать исчерпания возможностей системы.
- *Надежность*. Клиенты и сервисы могут использовать надежный протокол доставки сообщений для обеспечения гарантированной доставки, решения проблем с сетевым подключением и даже упорядочения вызовов.
- *Целостность*. Сервисы могут участвовать в одной единице работы, либо в транзакции (при поддержке со стороны инфраструктуры), либо в координируемой бизнес-транзакции, которая в конечном итоге должна быть целостной. Любая ошибка в цепочке вызовов приводит к отмене всего взаимодействия без привязки сервисов к природе ошибки и логике восстановления.
- *Синхронизация*. Вызовы сервиса могут автоматически синхронизироваться даже в том случае, если клиенты используют несколько параллельных потоков.

Типичные уровни

Метод рекомендует включить в архитектуру системы четыре уровня. Эти уровни соответствуют классическим практикам проектирования программных продуктов. Впрочем, применение нестабильности для управления декомпозицией внутри этих уровней может быть чем-то новым для вас. На рис. 3.4 изображены типичные уровни Метода.

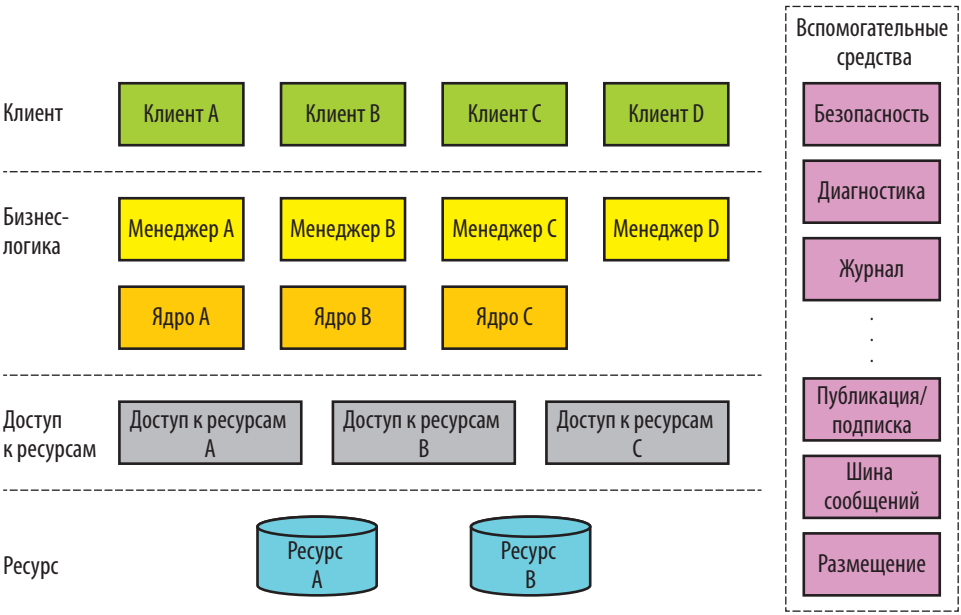


Рис. 3.4. Типичные уровни при применении Метода

Клиентский уровень

На вершине архитектуры находится *клиентский уровень*, также называемый *презентационным уровнем*. Мне последний термин кажется несколько неточным. Он подразумевает представление некоторой информации для пользователей, словно это все, что ожидается от верхнего уровня. Элементы клиентского уровня вполне могут быть приложениями для конечного пользователя, но они также вполне могут быть другими системами, взаимодействующими с вашей системой. Это весьма важное различие: обращаясь с вызовом к клиентскому уровню, вы уравниваете всех возможных клиентов и рассматриваете их по одним правилам. Все *Клиенты* (будь то приложения для конечного пользователя или другие системы) используют одни и те же точки входа в систему (что является важным аспектом любого хорошего решения), и на них распространяются единые требования к безопасности доступа, типов данных и к другим взаимодействиям. В свою очередь, это улучшает возможности повторного использования и расширяемость, а также упрощает сопровождение, так как исправление в одной точке входа распространяется на все *Клиенты*.

Потребление сервисов *Клиентами* способствует лучшему разделению представления и бизнес-логики. Многие сервисно-ориентированные технологии чрезвычайно строго относятся к типам данных, которые могут проходить через конечные точки. Тем самым ограничивается возможность привязки *Клиентов*

к сервисам, все *Клиенты* рассматриваются по единым правилам, а добавление новых типов *Клиентов* (по крайней мере в теории) осуществляется проще.

Клиентский уровень также инкапсулирует потенциальную нестабильность в клиентах. В вашей системе на текущий момент и в будущем по осям нестабильности могут существовать разные *Клиенты*: настольные приложения, веб-порталы, мобильные приложения, голограммы и дополненная реальность, API, административные приложения и т. д. Различные клиентские приложения будут использовать разные технологии, будут развертываться по разным правилам, иметь собственные версии и жизненные циклы и могут разрабатываться разными командами. В самом деле, клиентский уровень часто оказывается самой нестабильной частью типичной программной системы. Однако вся эта нестабильность инкапсулируется в разных блоках клиентского уровня, а изменения в одном компоненте не влияют на другой клиентский компонент.

Уровень бизнес-логики

Уровень бизнес-логики инкапсулирует нестабильность в бизнес-логике системы. Этот уровень реализует требуемое поведение системы, которое, как упоминалось ранее, лучше всего выражается в сценариях использования. Если бы сценарии были статическими, то в уровне бизнес-логики не было бы необходимости. Однако сценарии использования нестабильны по отношению как к заказчикам, так и ко времени. Так как сценарий использования содержит последовательность активностей в системе, конкретный сценарий использования может изменяться только в двух направлениях: либо изменяется сама последовательность, либо изменяется содержание отдельных сценариев. Например, сравните сценарий использования на рис. 3.1 со сценариями на рис. 3.5.

Во всех четырех случаях использования на рис. 3.1 и 3.5 задействованы одинаковые активности A, B и C, но каждая последовательность уникальна. Ключевой момент заключается в том, что последовательность, или координация процесса, может изменяться независимо от активностей.

Теперь возьмем две диаграммы активности на рис. 3.6. На обеих диаграммах показана одна последовательность, но с разными активностями. Активности могут изменяться независимо от последовательности.

Как последовательность, так и активности нестабильны, и в Методе эти нестабильности инкапсулируются в конкретных компонентах, которые называются *Менеджерами* и *Ядрами*. *Менеджеры* инкапсулируют нестабильность в последовательности, тогда как *Ядра* инкапсулируют нестабильность в активности. В примере декомпозиции системы биржевой торговли из главы 2 компонент *Поток операций торговли* (рис. 2.13) является *Менеджером*, а компонент *Преобразование данных — Ядром*.

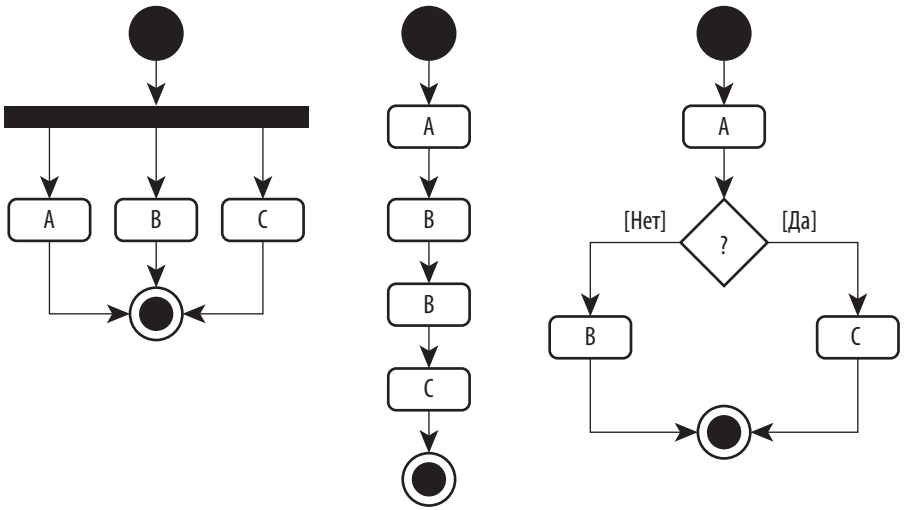


Рис. 3.5. Нестабильность последовательности

Поскольку сценарии использования часто связаны между собой, в *Менеджерах* обычно инкапсулируются семейства логически связанных сценариев использования (например, относящихся к конкретной подсистеме). Например, в торговой системе из главы 2 компонент *Поток операций анализа* является *Менеджером*, отличным от компонента *Поток операций торговли*, а каждый *Менеджер* также имеет сопутствующий набор сценариев использования для выполнения. *Ядра* имеют более ограниченную область действия, в них инкапсулируются бизнес-правила и активности.

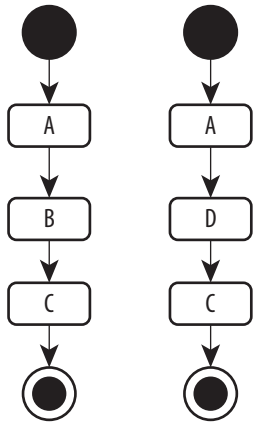


Рис. 3.6. Нестабильность активности

Так как последовательность может содержать значительную нестабильность без какой-либо нестабильности в активностях последовательности (рис. 3.5), менеджеры могут использовать 0 и более *Ядер*. *Ядра* могут совместно использоваться несколькими *Менеджерами*, потому что вы можете выполнить активность в одном сценарии использования по поручению одного *Менеджера*, а затем выполнить ту же активность для другого *Менеджера* в отдельном сценарии использования. *Ядра* следует проектировать с расчетом на повторное использование. Тем не менее если два *Менеджера* используют два разных *Ядра* для выполнения одной активности, вы либо имеете дело с функциональной декомпозицией, либо упустили какую-то нестабильность в активности. Мы еще вернемся к *Менеджерам* и *Ядрам* позднее в этой главе.

Уровень доступа к ресурсам

Уровень доступа к ресурсам инкапсулирует нестабильность обращения к ресурсу, а компонентам этого уровня присваивается имя *Доступ к ресурсу*. Например, если ресурс представляет собой базу данных, существуют буквально десятки разных способов обращения к базе данных, и ни один метод не будет лучше всех остальных методов во всех отношениях. Возможно, со временем вы захотите изменить способ обращения к базе данных, поэтому такое изменение или связанная с ним нестабильность должны быть инкапсулированы. Учтите, что простой инкапсуляции обращения к ресурсу недостаточно; вы также должны инкапсулировать нестабильность в самом ресурсе (локальная или облачная база данных, хранение в памяти или постоянное хранение). Изменения ресурсов также безусловно приводят к изменению компонентов *Доступ к ресурсу*.

Хотя мотивация за уровнем доступа к ресурсам сразу очевидна, а многие системы содержат некоторую разновидность уровня доступа, большинство таких уровней в конечном итоге для инкапсуляции нижележащей нестабильности создают контракт *Доступ к ресурсу*, напоминающий операции ввода/вывода в стиле CRUD. Например, если контракт вашего сервиса *Доступ к ресурсу* содержит такие операции, как `Select()`, `Insert()` и `Delete()`, то, скорее всего, ресурсом является база данных. Если позднее база данных будет заменена распространенной хеш-таблицей на облачной платформе, этот контракт станет бесполезным, и потребуется новый контракт. Изменение контракта повлияет на каждое *Ядро* и каждый *Менеджер*, использующий компонент *Доступ к ресурсу*. Аналогичным образом следует избегать таких операций, как `Open()`, `Close()`, `Seek()`, `Read()` и `Write()`, которые показывают, что нижележащий ресурс представляет собой файл. Хорошо спроектированный компонент *Доступ к ресурсу* в своем контракте предоставляет атомарные бизнес-команды для работы с ресурсом.

Использование атомарных бизнес-команд

Сервисы-менеджеры в системе выполняют некую последовательность бизнес-активностей. В свою очередь, эти активности часто представляют собой наборы еще более детализированных активностей. Однако в какой-то момент активности станут настолько низкоуровневыми, что их нельзя будет выразить любыми другими активностями в системе. В терминологии Метода эти неделимые активности называются атомарными бизнес-командами. Например, в банке классическим сценарием использования будет перевод средств между двумя счетами. Перевод осуществляется начислением средств на один счет и списанием их с другого счета. В банке начисление и списание средств являются атомарными операциями с точки зрения бизнеса. Учтите, что с точки зрения системы реализация атомарной бизнес-команды может состоять из нескольких шагов. Атомарность относится к бизнесу, а не к системе.

Атомарные бизнес-команды практически неизменяемы, потому что они сильно связаны с природой бизнеса, которая, как упоминалось в главе 2, практически никогда не изменяется. Например, банки выполняют операции начисления и списания средств еще со времен Медичи. Во внутренней реализации сервис *Доступ к ресурсу* должен преобразовывать эти команды из своего контракта в операции CRUD или операции ввода/вывода с ресурсами. Открывая доступ только к стабильным бизнес-командам, при изменении сервиса *Доступ к ресурсу* изменится только внутреннее строение компонента доступа, а не вся система поверх него.

Повторное использование компонентов Доступ к ресурсу

Сервисы *Доступ к ресурсу* могут совместно использоваться *Менеджерами* и *Ядрами*. Вы должны явно проектировать компоненты *Доступ к ресурсу* с учетом возможности повторного использования. Если два *Менеджера* или два *Ядра* не могут использовать один сервис *Доступ к ресурсу* при обращении к одному ресурсу или нуждаются в специфическом доступе, возможно, вы не инкапсулировали некоторую нестабильность в доступе или не смогли правильно изолировать атомарные бизнес-команды.

Уровень ресурсов

На уровне ресурсов находятся физические ресурсы, от которых зависит работа системы: базы данных, файловая система, кэш или очередь сообщений. В Методе *Ресурс* может быть внутренним и внешним по отношению к системе. Часто *Ресурс* представляет собой целую систему, но с точки зрения вашей системы он выглядит как обычный *Ресурс*.

Вспомогательные средства

На вертикальной панели в правой части рис. 3.4 перечислены вспомогательные сервисы. Они образуют своего рода общую инфраструктуру, которая необходима для работы почти всех систем. К числу таких вспомогательных сервисов относятся средства безопасности, регистрации данных в журнале, диагностики, публикации/подписки, шины сообщений, размещения и т. д. Позднее в этой главе будет показано, что для *Вспомогательных средств* нужны особые правила — не такие, как для других компонентов.

Рекомендации по классификации

Как и любая хорошая идея, Метод может стать предметом злоупотреблений. Без практики и критического мышления можно формально пользоваться

только таксономией Метода — и при этом все равно прийти к функциональной декомпозиции. Этот риск можно до определенной степени сократить, следуя простым рекомендациям из этого раздела.

Еще одно возможное применение для этих рекомендаций — начальная стадия проектирования. В начале практически любой деятельности по проектированию многие люди находятся в замешательстве и обычно даже не знают, с чего начать. Очень полезно вооружиться некоторыми ключевыми наблюдениями, которые помогут как запустить, так и проверить зарождающееся решение на жизнеспособность.

«Что в имени тебе моем»

Имена сервисов, как и диаграммы, играют важную роль в распространении вашего решения среди других участников. Содержательные имена играют настолько важную роль на уровнях бизнес-логики и доступа к ресурсам, что Метод рекомендует применять следующие соглашения при их выборе:

- Имена сервисов должны состоять из двух слов, записанных по правилам регистра языка Pascal.
- Суффикс имени всегда должен соответствовать типу сервиса — например, `Manager`, `Engine` или `Access` (в `ResourceAccess`).
- Префикс зависит от типа сервиса:
  - Для *Менеджеров* (`Manager`) префикс должен представлять собой существительное, связанное с инкапсулируемой нестабильностью в сценариях использования.
  - Для *Ядер* (`Engine`) префикс должен представлять собой существительное, описывающее инкапсулируемую активность.
  - Для компонентов *Доступ к ресурсам* (`ResourceAccess`) префикс должен представлять собой существительное, связанное с ресурсом, — например, данные, которые сервис предоставляет потребляющим сценариям использования.
  - Атомарные бизнес-команды не должны использоваться в качестве префикса в именах сервисов. Их применение должно ограничиваться именами операций в контрактах взаимодействия с уровнем доступа к ресурсам.

Например, в банковской системе `AccountManager` и `AccountAccess` могут быть приемлемыми именами сервисов. С другой стороны, имена `BillingManager` и `BillingAccess` отдают функциональной декомпозицией, потому что префикс-герундий с окончанием «ing» ассоциируется с «выполнением» чего-то вместо координации или нестабильности доступа. `CalculatingEngine` — хорошее воз-

можное имя, потому что *Ядра* «выполняют» такие операции, как агрегирование, адаптация, проверка на действительность, оценка, вычисление, преобразование, генерирование, регулирование, перевод и поиск. С другой стороны, имя `AccountEngine` лишено каких-либо признаков нестабильности доступа и снова сильно отдает функциональной декомпозицией или декомпозицией предметной области.

Четыре вопроса

Уровни сервисов и ресурсов в архитектуре приближенно соответствуют четырем вопросам: «кто», «что», «как» и «где». «Кто» взаимодействует с системой — на клиентском уровне, «что» требуется от системы — в менеджерах, «как» система выполняет бизнес-операции — в ядрах, «как» система обращается к ресурсам — на уровне доступа к ресурсам и «где» хранится состояние системы — на уровне ресурсов (рис. 3.7).

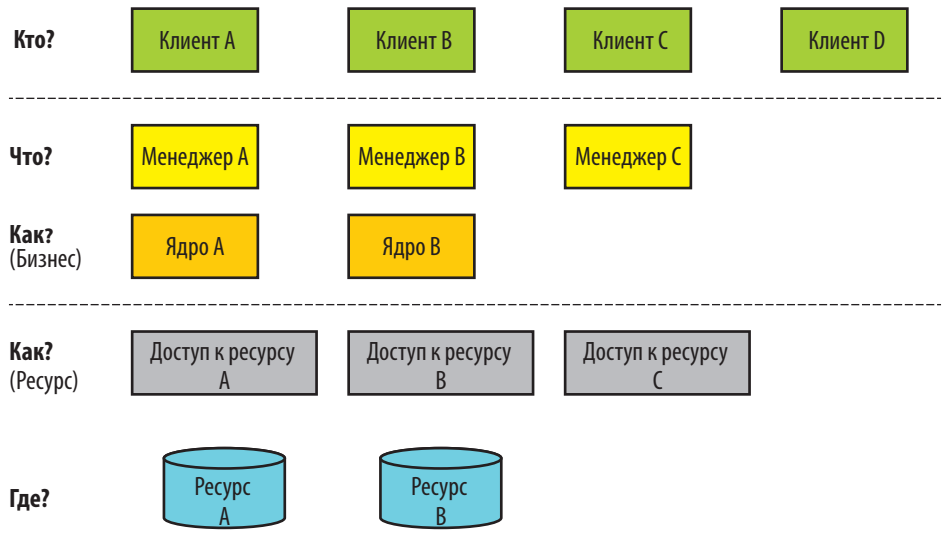


Рис. 3.7. Вопросы и уровни

Четыре вопроса приближенно соответствуют уровням, потому что нестабильность выходит на первый план. Например, если в вопросе «как» нестабильность минимальна или вовсе отсутствует, *Менеджеры* могут заниматься вопросами «что» и «как».

Задавать себе эти четыре вопроса и отвечать на них полезно в обеих критических точках работы по проектированию, при запуске и для проверки данных. Если вы начинаете с пустого места и не имеете четкого представления о том,

с чего начинать, можно начать с ответов на четыре вопроса. Постройте список всех «кто» и поместите их в одну группу как кандидатов на роль *Клиентов*. Постройте список всех «что» и поместите их в другую группу как кандидатов на роль *Менеджеров*, и т. д. Результат не будет идеальным; например, все компоненты «что» не обязательно преобразуются в отдельных *Менеджеров*, но это только начало.

После того как проектирование будет завершено, отступите на шаг и изучите результат. Все ли ваши *Клиенты* относятся к категории «кто» без малейшего следа «что»? Все ли *Менеджеры* относятся к категории «что» и у них нет примеси «кто» и «где»? И снова соответствие между вопросами и уровнями не будет идеальным. В некоторых случаях между вопросами могут возникнуть пересечения. Но если вы убеждены, что инкапсуляция нестабильности оправданна, нет причин далее сомневаться в этом выборе. Если же вы не убеждены, то вопросы могут стать своего рода «красным светом» — возможно, к вашей декомпозиции стоит присмотреться повнимательнее.

Соотношение Менеджеров и Ядер

В большинстве проектировочных решений *Ядер* оказывается меньше, чем можно было бы представить изначально. Прежде всего, чтобы в архитектуре существовало *Ядро*, должна существовать некоторая фундаментальная нестабильность, которую необходимо инкапсулировать, то есть неизвестное количество способов решения некоторой задачи. Такие нестабильности встречаются нечасто. Если ваше решение содержит большое количество *Ядер*, возможно, вы непреднамеренно провели функциональную декомпозицию.

Во время работы в IDesign мы на примере многочисленных систем сделали вывод о том, что между *Менеджерами* и *Ядрами* обычно существует оптимальное соотношение. Если в вашей системе существует только один *Менеджер* (не «всемогущий сервис»!), в ней либо вообще нет *Ядер*, либо присутствует не более одного *Ядра*. На происходящее можно взглянуть так: если система настолько проста, что в ней хватает всего одного приличного *Менеджера*, насколько вероятно иметь высокую нестабильность в активностях без такого же количества типов сценариев использования?

Как правило, в системах с двумя *Менеджерами* обычно используется одно *Ядро*. Если в системе присутствуют три *Менеджера*, то с большой вероятностью два *Ядра* будут оптимальны. Если в системе присутствуют пять *Менеджеров*, то может потребоваться до трех *Ядер*. Если же в системе восемь *Менеджеров*, то вам уже не удалось разработать хорошую архитектуру: большое количество *Менеджеров* — верный признак функциональной декомпозиции или декомпозиции предметной области. В большинстве систем никогда не будет такого количества *Менеджеров*, потому что в них не будет действительно независимых семейств сценариев использования с собственной нестабильно-

стью. Кроме того, *Менеджер* сможет поддерживать сразу несколько семейств сценариев использования, часто выражаемых различными контрактами сервисов или *гранями* (facets) одного сервиса. Это может привести к дальнейшему сокращению количества *Менеджеров* в системе.

Ключевые наблюдения

Вооружившись рекомендациями Метода, можно сделать некоторые обобщающие наблюдения относительно качеств, присущих хорошо спроектированной системе. Отклонения от этих наблюдений могут указывать на хроническую функциональную декомпозицию или, по крайней мере, на скороспелую декомпозицию, в которой вы инкапсулировали некоторые очевидные нестабильности, но упустили из виду другие.

> **ПРИМЕЧАНИЕ** Четко определенная терминология для различных аспектов архитектуры открывает возможность для передачи подобных наблюдений и рекомендаций.

Убывание нестабильности сверху вниз

В хорошо спроектированной системе нестабильность должна убывать сверху вниз между уровнями. *Клиенты* чрезвычайно нестабильны. Одни заказчики хотят, чтобы их *Клиенты* были такими, другие заказчики хотят, чтобы они были другими, а третьи хотят того же самого, но на другом устройстве. Этот естественный высокий уровень нестабильности не имеет никакого отношения к требуемому поведению системы. *Менеджеры* изменяются, но не в такой степени, как их *Клиенты*. *Менеджеры* изменяются при изменении сценариев использования (то есть требуемого поведения системы). *Ядра* обладают меньшей нестабильностью, чем *Менеджеры*. Чтобы *Ядро* изменилось, ваш бизнес должен изменить свой способ выполнения некоторой активности, а это происходит реже, чем изменение упорядочения активностей. Сервисы *Доступ к ресурсу* обладают еще меньшей нестабильностью, чем *Ядра*. Часто ли вы изменяете способ обращения к *Ресурсу* или, если на то пошло, изменяете сам *Ресурс*?

Активности и их последовательность можно изменять без изменения соответствий атомарных бизнес-команд и *Ресурсов*. *Ресурсы* являются наименее нестабильными компонентами, которые изменяются очень медленно по сравнению с остальными аспектами системы.

Проектировочное решение, в котором нестабильность убывает по уровням, обладает чрезвычайно высокой ценностью. От компонентов нижних уровней зависит больше элементов. Если компоненты, от которых вы зависите в наибольшей степени, оказываются наиболее нестабильными, то ваша система развалится.

Возможность повторного использования возрастает сверху вниз

Возможность повторного использования, в отличие от нестабильности, должна возрастать при переходе сверху вниз по уровням. *Клиенты* вряд ли хоть когда-нибудь пригодны для повторного использования. Клиентское приложение обычно разрабатывается для конкретной платформы и рынка, что не позволяет использовать его повторно. Например, код веб-портала не может быть легко использован в настольном приложении, а код настольного приложения — на мобильном устройстве. *Менеджеры* могут использоваться повторно, потому что один *Менеджер* и сценарии использования могут использоваться для разных *Клиентов*. *Ядра* обладают еще большим потенциалом повторного использования, чем *Менеджеры*, потому что одно ядро может вызываться несколькими *Менеджерами* в разных сценариях использования для выполнения одной активности. Компоненты *Доступ к ресурсу* в значительной мере пригодны для повторного использования, потому что они могут вызываться *Ядрами* и *Менеджерами*. *Ресурсы* являются наиболее пригодным для повторного использования элементом в любой прилично спроектированной системе. Способность повторного использования существующих *Ресурсов* в новых проектах часто становится ключевым фактором одобрения реализации новой системы со стороны бизнеса.

Почти расходные Менеджеры

Менеджеры делятся на три категории: дорогостоящие, расходные и почти расходные. О том, к какой категории принадлежит *Менеджер*, можно судить по тому, как вы будете реагировать на предложение об его изменении. Если вы будете всячески сопротивляться изменениям, опасаться сопряженных с ними затрат, возражать против изменения и т. д., то *Менеджер* очевидным образом является дорогостоящим. Это означает, что *Менеджер* слишком велик — вероятно, из-за функциональной декомпозиции. Если на запрос об изменениях вы без особых раздумий отправляете *Менеджера* в утиль, значит, *Менеджера* можно считать расходным. Расходные *Менеджеры* почти всегда свидетельствуют о дефектах проектирования и нарушении архитектуры. Они часто существуют только для того, чтобы удовлетворять рекомендации по проектированию без реальной необходимости инкапсуляции нестабильности сценариев использования.

Впрочем, если на предлагаемое изменение *Менеджера* вы начинаете размышлять, продумывая различные способы адаптации *Менеджера* к изменению сценария использования (и возможно, даже быстро оценивая объем необходимой работы), *Менеджер* является почти расходным. Если *Менеджер* просто координирует ядра и компоненты *Доступ к ресурсу*, инкапсулируя не-

стабильность последовательности, вы спроектировали отличный *Менеджер*. Хорошо спроектированный сервис *Менеджер* практически всегда является почти расходным.

Подсистемы и сервисы

Менеджеры, *Ядра* и компоненты *Доступ к ресурсу* сами по себе являются сервисами. Плотное взаимодействие между *Менеджерами*, *Ресурсами* и компонентами *Доступ к ресурсу* могут составлять один логический сервис для внешнего потребителя. Такой набор взаимодействующих сервисов может рассматриваться как логическая подсистема. Они группируются в вертикальный сегмент системы (рис. 3.8), при этом каждый вертикальный сегмент реализует соответствующий набор сценариев использования.

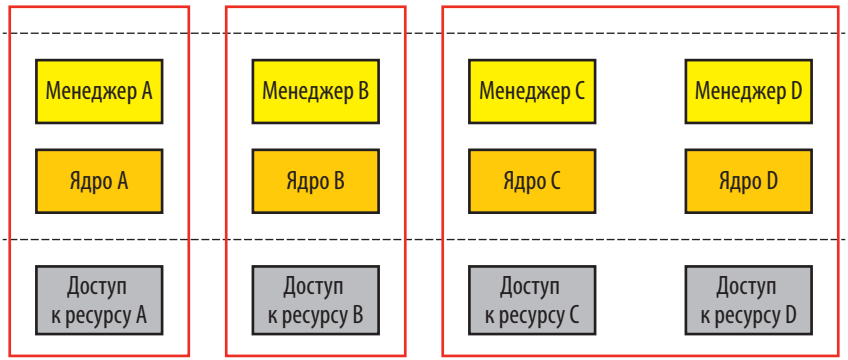


Рис. 3.8. Подсистемы как вертикальные сегменты

Избегайте чрезмерного разбиения вашей системы на подсистемы. В большинстве систем количество подсистем достаточно мало. Аналогичным образом следует ограничивать количество *Менеджеров* на подсистему — их должно быть не более трех. Это также позволяет вам несколько увеличить общее количество *Менеджеров* в системе по всем подсистемам.

Инкрементное построение

Если система относительно проста и невелика, то бизнес-ценность системы, то есть результат выполнения сценариев использования, с большой вероятностью потребует всех компонентов архитектуры. Для таких систем нет смысла публиковать, скажем, только *Ядра* или компоненты *Доступ к ресурсу*.

В большой системе некоторые подсистемы (например, вертикальные сегменты на рис. 3.8) могут стоять отдельно и предоставлять прямую бизнес-ценность. Построение таких систем обходится дороже и занимает больше времени. В таких случаях есть смысл разрабатывать и поставлять систему по этапам — по одному сегменту, в отличие от публикации одного выпуска в конце проекта. Более того, при инкрементных выпусках заказчик сможет предоставить раннюю обратную связь разработчикам.

Как с большими, так и с малыми системами при построении следует руководствоваться другим универсальным принципом:

Проектируйте по итерациям, стройте инкрементно.

Принцип остается истинным независимо от предметной области и отрасли. Представьте, что вы хотите построить дом на купленном участке земли. Даже лучший архитектор не сможет спроектировать дом за один сеанс. Неизбежно вы будете снова и снова возвращаться к определению задачи и обсуждению ограничений (бюджет, жильцы, стиль, время и риск). Все начнется с первых набросков чертежей, их проработки, оценки последствий и изучения альтернатив. После нескольких итераций решение начнет сходиться к итоговой точке. Когда придет время строить дом, будете ли вы это делать по итерациям? Начнете ли вы с палатки на двух человек, расширите ее до палатки на четырех человек, потом до хижины, до маленького домика и, наконец, до дома нужных размеров? Даже допускать такую возможность было бы безумием.

Скорее всего, вместо этого вы зальете фундамент, затем воздвигнете стены первого этажа, подключите коммуникации, добавите второй этаж и в итоге возведете крышу. Короче говоря, простой дом строится в инкрементном режиме. Для потенциального домовладельца один фундамент или крыша не обладают ценностью. Иначе говоря, дом, как и инкрементно построенная простая программная система, не обладает реальной ценностью до завершения. Тем не менее если здание состоит из нескольких этажей (или нескольких флигелей), его можно строить в инкрементном режиме и предоставлять промежуточную ценность. Возможно, ваш архитектор позволит строить здание по этажам (или по флигелям) по аналогии с подходом «по одному сегменту» в больших программных системах.

Другой пример — сборка автомобилей. Хотя в компании может быть группа проектировщиков, проектирующая машину между итерациями, когда наступает момент для сборки машины, процесс производства не начинается со скейтборда, расширяется до самоката, затем велосипеда, мотоцикла и, наконец, до автомобиля. Вместо этого машина собирается в инкрементном режиме. Сначала рабочие сваривают раму, затем закрепляют блок двигателя, добавляют си-

денья и шины. Они красят машину, добавляют приборную панель и, наконец, добавляют обивку сидений.

Существуют две причины, по которым система должна строиться инкрементно, а не итеративно. Во-первых, итеративное построение ужасно расточительно и сложно (переделать мотоцикл в автомобиль намного сложнее, чем просто собрать автомобиль). Во-вторых (что намного важнее), промежуточные итерации не обладают никакой бизнес-ценностью. Если заказчику нужна машина, чтобы отвозить детей в школу, что он будет делать с мотоциклом и почему он должен за мотоцикл платить?

Инкрементное построение также позволяет учитывать ограничения по времени и бюджету. Если вы проектируете четырехэтажный дом своей мечты, но можете позволить себе только одноэтажный дом, возможны два варианта. Первый — построить четырехэтажный дом в рамках одноэтажного бюджета: фанера вместо стен, листы пластика вместо окон, ведро вместо туалета, грязь на этажах и соломенная крыша. Второй вариант — нормально построить только первый этаж четырехэтажного дома. Когда у вас появятся дополнительные средства, можно будет перейти ко второму и третьему этажу. Через 10 лет, когда вы завершите строительство, эта конструкция все равно будет соответствовать исходной архитектуре.

Способность инкрементального построения по времени в рамках основана на том, что архитектура остается постоянной и истинной. При функциональной декомпозиции вы сталкиваетесь с постоянно перемещающейся кучей мусора. Разумно предположить, что те, кто владеет только функциональной декомпозицией, обречен на итеративное построение. При декомпозиции, основанной на нестабильности, у вас появляется шанс сделать все правильно.

Расширяемость

Вертикальные сегменты системы также позволяют обеспечить расширяемость. Правильный способ расширения любой системы не сводится к тому, чтобы вскрыть систему и бить кувалдой по существующим компонентам. Если система была правильно спроектирована с расчетом на расширяемость, вы можете легко оставить существующие аспекты без изменения и расширять систему в целом. Продолжая аналогию с домом, если когда-нибудь в будущем вы захотите добавить второй этаж к одноэтажному дому, то первый этаж должен быть спроектирован с расчетом на дополнительную нагрузку, водопровод должен быть проложен с возможностью расширения на второй этаж и т. д. Добавление второго этажа путем уничтожения первого этажа и последующим строительством первого и второго этажей называется переработкой, а не расширением. Проектирование системы на основе Метода ориентировано на расширяемости: просто добавьте новые сегменты или подсистемы.

О микросервисах

Меня называют одним из первопроходцев в области микросервисов. Еще в 2006 году в своих докладах и статьях я призывал строить системы, в которых каждый класс является сервисом.[1,2] Это требует применения технологии, которая может поддерживать такое детализированное использование сервисов. Я расширил WCF (Windows Communication Foundation) для этой цели: каждый класс можно взять и интерпретировать как сервис с сохранением традиционной модели программирования классов.[3] Я никогда не называл эти сервисы «микросервисами». Тогда я не знал о существовании концепции микросервисов — как, впрочем, отрицаю их и сейчас. Никаких микросервисов нет — есть только сервисы. Например, водяной насос в моей машине предоставляет критический сервис, а его длина составляет всего 20 сантиметров. Водяной насос, который местная водопроводная компания использует для подачи воды в мой город, выполняет очень полезную функцию для города, но его длина превышает 2,5 метра. Оттого, что где-то существует насос больших размеров, насос в моей машине не становится микронасосом: он все равно остается насосом. Сервисы остаются сервисами независимо от размера. Чтобы понять происхождение искусственной концепции микросервисов, необходимо вспомнить историю сервисно-ориентированного программирования.

История и проблематика

На заре сервисно-ориентированного программирования в начале 2000-х годов многие организации просто открывали доступ к своей системе как единому целому в форме сервиса. Такой чудовищный монолит создавал непреодолимые сложности с сопровождением и расширением из-за его сложности. Через 10 лет мучений отрасль осознала ошибочность такого подхода и стала требовать более детализированных сервисов, которые получили название *микросервисов*. На практике микросервисы соответствуют доменам или подсистемам, то есть сегментам (выделены прямоугольниками) на рис. 3.8. У этой идеи в том виде, в каком она практикуется сегодня, есть три недостатка.

Первая проблема: подразумеваемое ограничение на количество сервисов. Если маленькие сервисы лучше больших, то почему мы останавливаемся на уровне подсистемы? Подсистема по-прежнему слишком велика как единица сервиса с наибольшей детализацией. Почему бы не сделать сервисами структурные элементы подсистем? Вы должны распространить преимущества сервисов по архитектуре как можно далее. В подсистеме Метода компоненты *Менеджер*, *Ядро* и *Доступ к ресурсу* в подсистеме также должны быть сервисами.

---

[1] https://ru.wikipedia.org/wiki/Микросервисная_архитектура#История

[2] Juval Löwy, Programming WCF Services, 1st ed. (O'Reilly Media, 2007), 543–553.

[3] Löwy, Programming WCF Services, 1st ed., pp. 48–51; Juval Löwy, Programming WCF Services, 3rd ed. (O'Reilly Media, 2010), 74–75.

ПРИМЕЧАНИЕ В приложении Б рассматривается проблема детализации сервиса. Приложение объясняет, почему слишком большое количество сервисов (соответствующих подсистемам) является признаком некачественного проектирования.

Вторая проблема — повсеместное использование функциональной декомпозиции в архитектурах микросервисов в отрасли в целом. Даже из-за одного этого фактора все проекты, основанные на микросервисах, обречены на неудачу. Специалистам, пытающимся строить микросервисы, придется сражаться со сложностями как функциональной декомпозиции, так и сервисно-ориентированного подхода, но без преимуществ, которые предоставляет модульность сервисов. Большинство проектов не выдержит этого двойного удара. Боюсь, микросервисы станут самым серьезным провалом в истории программирования. Удобные в сопровождении, пригодные для повторного использования, расширяемые сервисы возможны — но не при таком подходе.

Третья проблема связана с коммуникационными протоколами. Хотя выбор коммуникационных протоколов скорее относится к детализированному проектированию, нежели к архитектуре, о последствиях выбора стоит по крайней мере упомянуть. Подавляющее большинство стеков микросервисов (на момент написания книги) использует REST/WebAPI и HTTP для взаимодействия с сервисами. Многие создатели технологий и консультанты поощряют эту практику (возможно, потому, что повсеместное использование «наименьшего общего кратного» упрощает их жизнь). Однако эти протоколы разрабатывались для общедоступных сервисов, а не для шлюзов к системам. Как правило, ни одна хорошо спроектированная система не должна использовать одни и те же средства коммуникаций как во внутренних, так и во внешних механизмах.

Например, на моем ноутбуке имеется дисковый накопитель, который предоставляет очень важный сервис: хранение данных. Также ноутбук потребляет сервис, предоставляемый сетевым маршрутизатором для всех запросов DNS, и сервис сервера SMTP для работы с электронной почтой. Для внешних сервисов ноутбук использует TCP/IP; для внутренних (например, для чтения/записи данных на диск) используется интерфейс SATA. Ноутбук использует немало таких специализированных внутренних протоколов для выполнения своих основных функций.

Другой пример — человеческое тело. Печень обеспечивает очень важный сервис: обмен веществ. Ваше тело также предоставляет полезные сервисы вашим заказчикам и организации, а для общения с ними вы используете естественный язык (английский). Тем не менее вы не общаетесь со своей печенью на английском. Вместо этого для взаимодействия используются нервы и гормоны.

Протоколы, используемые для внешних сервисов, обычно работают медленно, с высокими затратами и высоким риском ошибок. Эти атрибуты указывают на высокую степень ослабления связей. Ненадежный протокол HTTP может

идеально подходить для внешних сервисов, но он нежелателен для внутренних сервисов, у которых коммуникации и сервисы должны быть безупречными.

Ошибочный выбор протокола между сервисами может иметь фатальные последствия. Если вы не поговорили со своим начальником или не достигли полного взаимопонимания с заказчиком, это еще не конец света, но если вы не сможете правильно (или вообще как-нибудь) взаимодействовать со своей печенью, вы умрете.

Схожие проблемы существуют в отношении специализации и эффективности. Использование HTTP между внутренними сервисами напоминает попытки управлять внутренними сервисами вашего тела на английском языке. Даже если слова идеально четко произнесены и понятны, английский язык не обладает адаптируемостью, быстродействием и терминологией, необходимой для описания взаимодействий между внутренними сервисами.

Внутренние сервисы, такие как *Ядра* и компоненты *Доступ к ресурсам*, зависят от быстрых, надежных, высокопроизводительных каналов связи. К их числу относятся TCP/IP, именованные каналы, IPC, доменные сокеты, специализированные цепочки перехвата в памяти, очереди сообщений и т. д.

Открытые и закрытые архитектуры

Любая многоуровневая архитектура может иметь одну из двух возможных операционных моделей: *открытую* или *закрытую*. В этом разделе сравниваются две альтернативы. Из этого обсуждения можно почерпнуть некоторые рекомендации по проектированию в контексте классификации сервисов.

Открытая архитектура

В открытой архитектуре любой компонент может обращаться с вызовом к любому другому компоненту независимо от того, на каких уровнях эти компоненты находятся. Вызовы могут быть восходящими, нисходящими, горизонтальными — как считаете нужным. Открытые архитектуры обладают наибольшей гибкостью. Тем не менее в открытой архитектуре эта гибкость достигается за счет нарушения инкапсуляции и усиления связей.

Например, представьте, что на рис. 3.4 *Ядра* напрямую обращаются с вызовами к *Ресурсам*. Хотя такой вызов технически возможен, представьте, что вы захотели заменить *Ресурсы* или просто изменить способ обращения к ним. Внезапно выясняется, что все *Ядра* тоже должны измениться. Как насчет того, чтобы клиенты вызывали сервисы *Доступ к ресурсу* напрямую? Хотя этот вариант не так плох, как прямые вызовы самих *Ресурсов*, вся бизнес-логика должна мигрировать в *Клиентов*. В этом случае при любых изменениях в бизнес-логике придется перерабатывать *Клиенты*.

Вызовы к верхним уровням также нежелательны. Что, если на рис. 3.4 *Менеджер* вызовет *Клиент*, чтобы обновить некий элемент пользовательского интерфейса? Теперь при изменении пользовательского интерфейса *Менеджер* должен будет реагировать и на это изменение. Таким образом, нестабильность *Клиентов* была импортирована в *Менеджеры*.

Горизонтальные вызовы (внутриуровневые) также создают непропорционально высокую связанность. Представьте, что *Менеджер A* вызывает *Менеджера B* на рис. 3.4. В данном случае *Менеджер B* — всего лишь активность внутри сценария использования, выполняемого *Менеджером A*. *Менеджер* должен инкапсулировать набор независимых сценариев использования. Выходит, сценарии использования *Менеджера B* теперь независимы от сценариев использования *Менеджера A*? Любые изменения выполнения активности *Менеджером B* нарушат работу *Менеджера A*, вызывая в памяти проблемы на рис. 2.5. Подобные горизонтальные вызовы почти всегда приводят к функциональной декомпозиции на уровне *Менеджеров*.

Как насчет *Ядра A*, которое вызывает *Ядро B*? Было ли *Ядро B* нестабильной активностью, которая существует отдельно от *Ядра A*? И снова за необходимостью сцепления вызовов ядер с большой вероятностью стоит функциональная декомпозиция.

Когда вы используете открытую архитектуру, само создание архитектурных уровней вряд ли предоставляет какие-либо преимущества. Как правило, в области разработки расплачиваться инкапсуляцией за гибкость — не лучший вариант.

Закрытая архитектура

Закрытая архитектура стремится довести до максимума преимущества многоуровневой архитектуры, для чего вызовы между уровнями и в пределах уровней запрещаются. Запрет на вызовы к нижним уровням обеспечит максимальное ослабление связей между этими уровнями, однако такое решение получится бесполезным. Закрытая архитектура открывает трещинку в уровнях, позволяя компонентам одного уровня вызывать компоненты соседнего нижнего уровня. Компоненты в пределах уровня могут предоставлять услуги компонентам уровня, находящегося непосредственно над ним, но они инкапсулируют то, что происходит на более низких уровнях. Закрытая архитектура способствует ослаблению связей, расплачиваясь гибкостью за инкапсуляцию. В общем случае этот компромисс более желателен, чем обратный.

Полузакрытая/полуоткрытая архитектура

Проблемы открытых архитектур — разрешение вызовов восходящих, нисходящих или горизонтальных — достаточно очевидны. Тем не менее все ли три греха

одинаково плохи? Худший вариант — восходящие вызовы: они не только создают межуровневое связывание, но и импортируют нестабильность более высокого уровня на нижние уровни. Второе по серьезности нарушение — горизонтальные вызовы, потому что они связывают компоненты внутри уровня. В закрытой архитектуре всегда разрешены вызовы на один уровень вниз, но как насчет вызова на несколько уровней вниз? Полузакрытая/полуоткрытая архитектура допускает вызов на несколько уровней вниз. Здесь гибкость и быстродействие снова достигаются за счет инкапсуляции, чего в общем случае делать не рекомендуется.

Использование полузакрытых/полуоткрытых архитектур оправданно в двух классических случаях. Первый встречается при проектировании ключевого блока инфраструктуры, из которого необходимо выжать каждую каплю быстродействия. В таких случаях последовательные переходы на несколько уровней вниз могут отрицательно отразиться на быстродействии. Для примера возьмем модель OSI (Open Systems Interconnection) сетевых коммуникаций, состоящую из семи уровней.[1] Когда разработчик реализует эту модель в своем стеке TCP, он не сможет позволить себе потерю быстродействия, накладываемую всеми семью уровнями при каждом вызове, и для него будет разумно выбрать для стека полузакрытую/полуоткрытую архитектуру. Второй случай встречается при работе с кодовыми базами, которые практически никогда не изменяются. Потеря инкапсуляции и дополнительное связывание в такой кодовой базе несущественны, потому что вам практически никогда не придется заниматься сопровождением этого кода. И снова реализация сетевого стека служит хорошим примером кода, который практически никогда не изменяется.

У полузакрытых/полуоткрытых архитектур тоже имеется полезное применение. Тем не менее многие системы не работают на уровне быстродействия, необходимом для оправдания таких решений, а их кодовые базы не остаются неизменными в такой степени.

Ослабление правил

В реальных бизнес-системах лучшим вариантом всегда остается закрытая архитектура. Обсуждение открытых и полуоткрытых архитектур должно отвратить вас от любого другого выбора.

Хотя закрытые архитектурные системы обладают самой слабой связанностью и наибольшей инкапсуляцией, для них также характерна наименьшая гибкость. Нехватка гибкости может привести к невероятным уровням сложности, обусловленным непрямыми обращениями и посредниками, а слишком жесткая архитектура нежелательна. Метод ослабляет правила закрытой архитектуры для сокращения сложности и непроизводительных затрат без вреда для инкапсуляции или связанности.

---

[1] https://ru.wikipedia.org/wiki/Сетевая_модель_OSI

Вызов вспомогательных средств

В закрытых архитектурах с вызовом *Вспомогательных средств* могут возникнуть трудности. Возьмем *Журнал* — сервис для протоколирования событий, возникающих во время выполнения. Если отнести *Журнал* к *Ресурсам*, то компоненты *Доступ к ресурсам* смогут пользоваться им, но для *Менеджеров* это будет невозможно. Если разместить *Журнал* на одном уровне с *Менеджерами*, то регистрировать события в *Журнале* смогут только клиенты. То же относится к *Безопасности* и *Диагностике* — сервисам, которые необходимы практически для всех других компонентов. Короче говоря, для *Вспомогательных средств* в закрытой архитектуре нет хорошего места. Метод размещает *Вспомогательные средства* на вертикальной панели сбоку от уровней (рис. 3.4). Панель проходит через все уровни, что позволяет любому компоненту в архитектуре использовать любое из *Вспомогательных средств*.

Некоторые разработчики пытаются злоупотреблять панелью *Вспомогательных средств*, нарекая *Вспомогательным средством* любой компонент, который хотят сделать легкодоступным на любом уровне. Не все компоненты могут находиться на этой панели. Чтобы компонент можно было отнести к *Вспомогательным средствам*, он должен пройти простую проверку: может ли этот компонент теоретически использоваться в любой другой системе — скажем, кофейном автомате? Например, кофейный автомат может использовать сервис *Безопасность* для проверки того, разрешено ли пользователю пить кофе. Также он может вести журнал того, сколько кофе пьют офисные работники, иметь систему диагностики и использовать сервис *Публикация/подписка* для публикации события, уведомляющего о нехватке кофе. Каждая из этих функций заслуживает инкапсуляции в сервисе *Вспомогательные средства*. С другой стороны, вам будет нелегко объяснить, зачем в набор *Вспомогательных средств* кофейного автомата включен сервис для расчета процентов по ипотеке.

Вызов компонентов Доступ к ресурсу из бизнес-логики

Следующая рекомендация может быть очевидной, но ее важно сформулировать в явном виде. Так как *Менеджеры* и *Ядра* находятся на одном уровне, они могут вызывать сервисы *Доступ к ресурсу* без нарушения закрытой архитектуры (рис. 3.4). Возможность вызова сервисов *Доступ к ресурсу* со стороны *Менеджеров* также следует из раздела, в котором приводятся определения *Менеджеров* и *Ядер*. *Менеджер*, не использующий *Ядра*, должен иметь возможность обращаться к нижележащим ресурсам.

Вызов ядер со стороны Менеджеров

Менеджеры могут напрямую вызывать *Ядра*. Разделение между *Менеджерами* и *Ядрами* происходит практически на уровне детализированного проектирования. *Ядра* в действительности всего лишь являются выражением паттерна

проектирования Стратегия,[1] используемого для реализации активностей в потоках операций *Менеджеров*. Таким образом, вызовы «*Менеджер-Ядро*» не являются горизонтальными в истинном смысле, в отличие от вызовов «*Менеджер-Менеджер*». Также можно рассматривать *Ядра* как находящиеся на другой плоскости, ортогональной к плоскости *Менеджеров*.

Очереди вызовов «Менеджер-Менеджер»

Хотя *Менеджеры* не должны обращаться с горизонтальными вызовами к другим *Менеджерам* напрямую, *Менеджер* может поставить в очередь вызов к другому *Менеджеру*. Существуют два возможных объяснения того, почему это не нарушает принцип закрытой архитектуры, — техническое и семантическое.

Техническое объяснение базируется на самой механике очереди вызовов. Когда *Клиент* вызывает сервис через очередь, он взаимодействует с посредником, который затем помещает сообщение в очередь сообщений для сервиса. Объект прослушивания очереди отслеживает состояние очереди, обнаруживает новое сообщение, извлекает его из очереди и вызывает сервис. При использовании структуры Метода, когда *Менеджер* ставит в очередь вызов к другому *Менеджеру*, посредником является сервис *Доступ к ресурсу* для используемого *Ресурса*, то есть очереди; таким образом, вызов в действительности идет сверху вниз, а не по горизонтали. Слушатель очереди, по сути, оказывается другим *Клиентом* в системе, и он тоже передает вызов сверху вниз *Менеджеру*-получателю. Никакие горизонтальные вызовы в действительности не выполняются.

В семантическом объяснении задействована природа сценариев использования. В бизнес-системах нередко присутствует один сценарий использования, который инициирует отложенное выполнение другого сценария. Например, представьте систему, в которой *Менеджер*, выполняющий сценарий использования, должен сохранить некоторое состояние системы для анализа в конце месяца. Без прерывания потока операций *Менеджер* может поставить в очередь запрос на анализ для другого *Менеджера*. Второй *Менеджер* может извлечь из очереди запрос в конце месяца и выполнить свой поток операций анализа. Два сценария использования независимы и изолированы друг от друга на временной оси.

Открытие архитектуры

Даже с самыми лучшими рекомендациями время от времени вам будут попадаться разработчики, пытающиеся открыть архитектуру горизонтальными вызовами, восходящими вызовами или другими нарушениями закрытой архитектуры. Не стоит с ходу отвергать эти нарушения или требовать слепого соответствия рекомендациям. Почти всегда такие отклонения указывают на неко-

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.

торую потребность, которая заставила разработчиков нарушить рекомендации. Вы должны правильно разрешить эту потребность способом, соответствующим принципу закрытой архитектуры. Допустим, во время проектирования или анализа кода вы обнаружили, что один *Менеджер* напрямую вызывает другого *Менеджера*. Разработчик может попытаться оправдать горизонтальный вызов, указав на некоторое требование другого сценария использования, которое должно выполняться как реакция на исходный сценарий. Однако крайне маловероятно, что реакция второго *Менеджера* должна была происходить немедленно. Постановка в очередь вызова между *Менеджерами* была бы признаком лучшей архитектуры и позволила избежать горизонтального вызова.

В другом анализе был обнаружен *Менеджер*, обращающийся с вызовом к *Клиенту*, — серьезное нарушение принципа закрытой архитектуры. В качестве оправдания разработчик указывает на требование об оповещении *Клиента* при выполнении некоторого условия. Хотя такое требование вполне действительно, восходящий вызов не является приемлемым решением. Со временем уведомления могут понадобиться другим *Клиентам* или у других *Менеджеров* может возникнуть необходимость в уведомлении *Клиента*. В данном случае вы обнаружили нестабильность в отношении того, кто уведомляет, и нестабильность в отношении получателя события. Эту нестабильность можно инкапсулировать при помощи сервиса *Публикация/подписка* на панели *Вспомогательных средств*. Конечно, *Менеджер* может обратиться с вызовом к этому *Вспомогательному средству*. В будущем добавление других *Клиентов*-подписчиков или *Менеджеров*, публикующих событие, станет тривиальной задачей и не будет иметь нежелательного эффекта для системы.

Запреты проектирования

При наличии определений как для сервисов, так и для уровней, также возможно составить список того, чего следует избегать, — своего рода «запретов проектирования». Некоторые элементы в списке могут показаться вам очевидными после предыдущих разделов, однако я видел их достаточно часто, чтобы заключить, что они вовсе не очевидны. Главная причина, по которой люди нарушают запреты, заключается в том, что они создали функциональную декомпозицию и убедили себя, что она не является функциональной.

Если вы включите один из пунктов списка в свою архитектуру, то, скорее всего, пожалеете об этом. Рассматривайте любое нарушение правил как тревожный сигнал и проведите дальнейшие исследования, чтобы понять, что вы упустили из виду:

- Клиенты не обращаются с вызовами к нескольким *Менеджерам* в одном сценарии использования. Такая ситуация указывала бы на то, что между *Менеджерами* существует тесная связь и они уже не представляют отдельные семейства сценариев использования, или отдельные подсистемы, или

отдельные сегменты. Сцепленные вызовы *Менеджеров* от *Клиентов* указывают на наличие функциональной декомпозиции и требуют от *Клиента* связать используемые функциональности воедино (рис. 2.1). Клиенты могут вызывать нескольких *Менеджеров*, но не в одном сценарии использования: например, *Клиент* может вызвать *Менеджера A* для выполнения сценария использования 1, а затем *Менеджера B* для выполнения сценария использования 2.

- *Клиенты* не обращаются с вызовами к *Ядрам*. Единственными точками входа на бизнес-уровень являются *Менеджеры*. *Менеджеры* представляют систему, а *Ядра* всего лишь являются внутренними подробностями реализации уровня. Если *Клиенты* обращаются с вызовами к *Ядрам*, то упорядочение сценариев использования и связанная с ним нестабильность вынуждены перемещаться в *Клиентов*, что приводит к загрязнению их бизнес-логикой. Вызовы *Ядер* из *Клиентов* — признак функциональной декомпозиции.

- *Менеджеры* не ставят в очередь вызовы более чем к одному *Менеджеру* в одном сценарии использования. Если два *Менеджера* получают вызов из очереди, то почему не третий? Почему не все остальные? Необходимость реагировать на вызов из очереди двум (и более) *Менеджерам* — убедительный признак того, что реагировать должны другие *Менеджеры* (а может быть, все), поэтому в такой ситуации следует использовать сервис *Публикация/подписка*.

- *Ядра* не получают вызовов из очередей. *Ядра* выполняют утилитарную функцию и существуют для выполнения нестабильной активности для *Менеджера*. Сами по себе они не имеют независимого смысла. Вызов в очереди по определению выполняется независимо от всех остальных компонентов в системе. Выполнение только активности *Ядра*, изолированной от каких-либо сценариев использования или других активностей, не имеет смысла в бизнес-контексте.

- Сервисы *Доступ к ресурсу* не получают вызовов из очереди. По аналогии с рекомендацией для *Ядер*, сервисы *Доступ к ресурсу* существуют для того, чтобы предоставлять услуги *Менеджерам* или *Ядрам*, и не обладают смыслом сами по себе. Независимое обращение к *Ресурсу* из любого другого компонента системы не имеет смысла в бизнес-контексте.

- *Клиенты* не публикуют события. События представляют изменения в состоянии системы, представляющие интерес для *Клиентов* (или *Менеджеров*). У *Клиента* нет необходимости уведомлять себя (или других клиентов). Кроме того, для выявления необходимости публикации события часто требуется знание внутреннего устройства системы — то знание, которого не должно быть у *Клиента*. Тем не менее при функциональной декомпозиции *Клиент* является системой, и он должен публиковать события.

- *Ядра* не публикуют события. Публикация событий требует обнаружения и реакции на изменения в системе; обычно она является одной из стадий

сценария использования, выполняемого *Менеджером*. *Ядро*, выполняющее активность, не может много знать о контексте активности или состоянии сценария использования.

- Сервисы *Доступ к ресурсу* не публикуют события. Сервисы *Доступ к ресурсу* не могут знать о важности состояния *Ресурса* для системы. Любая такая информация и поведение реакции должны находиться в *Менеджерах*.
- *Ресурсы* не публикуют события. Необходимость публикации события из *Ресурсов* часто является результатом сильно связанной функциональной декомпозиции. Как и в случае с *Доступом к ресурсу*, бизнес-логика такого рода должна находиться в *Менеджерах*. Когда *Менеджер* изменяет состояние системы, *Менеджер* также должен публиковать соответствующие события.
- *Ядра*, компоненты *Доступ к ресурсам* и *Ресурсы* не подписываются на события. Обработка события почти всегда является началом некоторого сценария использования, поэтому она должна выполняться в *Клиенте* или *Менеджере*. *Клиент* может уведомить пользователя о событии, а *Менеджер* может выполнить некоторое служебное поведение.
- *Ядра* никогда не вызывают друг друга. Такие вызовы не только нарушают принцип закрытой архитектуры, но и не имеют смысла в декомпозиции на основе нестабильности. *Ядро* уже должно инкапсулировать все, что относится к активности. Любые вызовы «*Ядро-Ядро*» являются признаком функциональной декомпозиции.
- Сервисы *Доступ к ресурсу* никогда не вызывают друг друга. Если сервисы *Доступ к ресурсу* инкапсулируют нестабильность атомарной бизнес-команды, одна атомарная команда не может требовать другой. Это правило похоже на правило о том, что *Ядра* не должны вызывать друг друга. Обратите внимание: однозначное соответствие между компонентами *Доступ к ресурсу* и *Ресурс* (каждый *Ресурс* имеет собственный компонент *Доступ к ресурсу*) не обязательно. Часто два и более *Ресурса* логически должны быть объединены для реализации некоторых атомарных бизнес-команд. Соединение должно осуществляться одним сервисом *Доступ к ресурсу* (вместо вызовов между сервисами *Доступ к ресурсу*).

Стремление к симметрии

Другое универсальное правило проектирования гласит, что все хорошие архитектуры симметричны. Представьте собственное тело. У вас нет третьей руки на правом боку, потому что эволюционные факторы были всесторонними и способствовали симметрии. Эволюционное воздействие также воздействует на программные системы, заставляя их реагировать на изменения окружения или вымереть. Однако стремление к симметрии проявляется только на уров-

не архитектуры, но не на уровне детализированного решения. Конечно, ваши внутренние органы не симметричны, потому что симметрия не предлагала никаких эволюционных преимуществ вашим предкам (то есть система умирает, когда вы открываете доступ к ее внутреннему устройству).

Симметрия в программных системах проявляется в закономерностях повторяющихся вызовов между сценариями использования. Вы должны ожидать симметрии, а ее отсутствие — причина для беспокойства. Например, представьте, что *Менеджер* реализует четыре сценария использования, три из которых публикуют событие при помощи сервиса *Публикация/подписка*, а четвертый — нет. Нарушения симметрии — плохой признак при проектировании. Почему четвертый сценарий выделяется на общем фоне? Что вы упускаете или, наоборот, делаете чрезмерно? Точно ли *Менеджер* является настоящим *Менеджером* или это компонент, полученный в ходе функциональной декомпозиции без нестабильности? Симметрия также может быть нарушена присутствием, а не только отсутствием чего-то. Например, если *Менеджер* реализует четыре сценария использования, из которых только один приводит к постановке в очередь вызова к другому *Менеджеру*, эта асимметрия также является плохим признаком. Симметрия — настолько фундаментальный признак качественного проектирования, что в общем случае одни и те же закономерности вызовов должны быть характерны для всех *Менеджеров*.

4 Композиция

Главная причина существования программных систем — способность удовлетворять интересы бизнеса за счет удовлетворения нужд и потребностей заказчика. В двух предыдущих главах было рассказано, как выполнить декомпозицию системы на компоненты для создания архитектуры. Декомпозиция на компоненты, по сути, определяет статическую структуру системы (вроде чертежа). На стадии выполнения система работает динамически, а разные компоненты взаимодействуют друг с другом. Но как определить, что композиция этих компонентов на стадии выполнения нормально удовлетворяет все требования? Проверка проектировочного решения связана с анализом требований, проектированием системы и вашей ценностью как архитектора. Как вскоре будет показано, проверка проектировочного решения и композиция тесно связаны. Вы можете и должны построить жизнеспособное решение и проверить его надежным, воспроизводимым образом.

Эта глава предоставит в ваше распоряжение инструменты для проверки того, что система не только решает текущие требования, но и может устоять перед будущими изменениями в требованиях. Для этого необходимо сначала понять природу требований и изменений и их связь с проектировочным решением системы. В свою очередь, это понимание приводит к фундаментальным наблюдениям о проектировании системы с практическими рекомендациями по производству действительного проектировочного решения.

Требования и изменения

Требования изменяются. Просто примите этот факт — такова природа требований.

Изменения требований порой бывают фантастическими. Если бы требования были статическими, все мы лишились бы работы: кто-то где-то в прошлом разработал адекватную версию системы, а система успешно работала бы с тех

пор. Чем сильнее изменяются требования, тем лучше техническим специалистам. Мир сильно зависит от программного обеспечения, но при этом в мире так мало разработчиков и архитекторов и так много других специалистов. Чем сильнее изменяются требования, тем выше спрос на услуги профессионалов, а поскольку запас профессионалов ограничен, их услуги оплачиваются выше.

Сопротивление изменениям

Изменения — это замечательно, но многие специалисты в течение всей своей карьеры сопротивляются таким изменениям. Причина проста: многие разработчики и архитекторы проектируют свою систему под конкретные требования. Они прикладывают значительные усилия к тому, чтобы преобразовать требования в компоненты архитектуры. Они стараются довести до максимума соответствие между требованиями и проектировочным решением. Однако при изменении требований их решение также должно измениться. В любой системе изменения в проектировочном решении приносят массу неприятностей, часто имеют разрушительные последствия и всегда обходятся дорого. Никто не любит неприятностей (пусть и причиняемых самому себе, как в данном случае), поэтому люди научились сопротивляться изменениям в требованиях — буквально отталкивая ту руку, которая их кормит.

Главная директива проектирования

Причина для такого расхождения с привычкой сопротивляться изменениям настолько проста, что она ускользает практически от всех специалистов на протяжении всех их карьер:

Никогда не проектируйте под конкретные требования.

Эта простая директива противоречит всему, чему вас учили и что вы усваивали на практике, хотя вроде бы она полностью очевидна. Любые попытки проектирования под конкретные требования всегда гарантируют неприятности. Так как неприятности — это плохо, ничто не оправдывает то, что ведет к столь нежелательным последствиям. Возможно, проектировщики даже отлично сознают, что их процесс проектирования не работает и никогда не работал, но при отсутствии альтернатив они прибегают к единственному выходу, который им известен, — к проектированию под конкретные требования.

> **ПРИМЕЧАНИЕ** Проблемы проектирования под конкретные требования не ограничиваются программными системами. В главе 2 рассматривается катастрофический опыт построения дома при проектировании под требуемую функциональность.

Бесполезность проектирования под конкретные требования

Как обсуждалось в главе 3, правильным способом отражения требований являются сценарии использования: требуемый набор аспектов поведения в системе. Система сколько-нибудь серьезного размера содержит десятки таких сценариев использования, а в больших системах их количество может исчисляться сотнями. В то же время никто в истории разработки не располагал временем для составления спецификаций сотен сценариев использования в начале работы над проектом.

Предположим, в первый день нового проекта вам вручают папку с 300 сценариями использования. Можно ли доверять правильности и полноте этой подборки? Удивит ли вас, если окажется, что на самом деле сценариев использования должно быть 330, а у вас не хватает части материала? Если вам выдадут 300 сценариев использования, удивит ли вас, если окажется, что на самом деле сценариев всего 200, потому что спецификация требований содержит много дубликатов? И если вы будете проектировать под конкретные требования в таком случае, не придется ли вам выполнить на 50% больше работы? Возможна ли ситуация, когда в полученном вами наборе некоторые сценарии использования являются взаимоисключающими? А как насчет риска дефективных сценариев, которые заставят вас реализовать ошибочное поведение?

Даже если кто-то чудом выделит приличное время, необходимое для отражения всех 300 сценариев использования на диаграммах активностей, чтобы полностью исключить отсутствие сценариев, согласовать взаимоисключающие сценарии и объединить дубликаты, эффективность такой работы будет небольшой, потому что требования все равно изменятся. Со временем появятся новые требования, некоторые существующие требования будут исключены, а другие требования просто изменятся «на месте». Короче, любые попытки собрать полный набор требований и проектировать под эти требования обречены на провал.

Компоновочное проектирование

Прежде чем описывать правильный подход к выполнению требований, необходимо правильно установить планку для выполнения требований. Целью любого проектирования системы должно быть выполнение *всех* сценариев использования. Слово «всех» в предыдущем предложении следует понимать буквально: текущих и будущих, известных и неизвестных сценариев использования. Планка устанавливается именно на этом уровне, ничто меньшее не подойдет. Если вы не сможете взять эту высоту, то в какой-то момент в будущем, когда изменятся требования, вашей архитектуре тоже придется изменяться. Верный признак некачественного проектирования — когда изменяются требования, проектировочное решение также приходится изменять.

Базовые сценарии использования

В любой конкретной системе не все сценарии использования уникальны. Многие сценарии использования являются разновидностями других сценариев. Главное требуемое поведение имеет множество модификаций — нормальный случай, неполный случай, случай для конкретного заказчика в конкретном локальном контексте, ошибочный случай и т. д. При этом существуют всего два типа сценариев использования: базовые сценарии использования и все остальные. Базовые сценарии использования представляют сущность бизнеса в системе. Как обсуждалось в главе 2, природа бизнеса почти никогда не изменяется, и базовые сценарии использования должны обладать тем же свойством. Конечно, обычные, небазовые сценарии использования будут изменяться достаточно часто и в зависимости от заказчика. Хотя ваши заказчики могут (и скорее всего, будут) иметь собственную интерпретацию и адаптацию обычных сценариев использования, все они разделяют одни и те же базовые сценарии использования.

Хотя система может иметь сотни сценариев использования, вас спасает то, что в ней будет лишь небольшая группа базовых сценариев. В своем опыте практической работы в IDesing мы обычно видим системы с крайне немногочисленными базовыми сценариями использования. В большинстве систем существуют всего два или три базовых сценария, их количество редко превышает шесть. Поразмыслите над своей системой, которой вы пользуетесь в офисе, или над недавним проектом, в котором вы участвовали; мысленно подсчитайте действительно уникальные сценарии использования, которые должна обрабатывать ваша система. Окажется, что это число мало, очень мало. Или попробуйте взять маркетинговую листовку с описанием системы и подсчитайте количество пунктов в маркированном списке. Скорее всего, их будет не более трех.

Поиск базовых сценариев использования

Базовые сценарии использования вряд ли будут явно выражены в документированных требованиях, каким бы подробным ни был этот документ. Их немногочисленность не означает, что базовые сценарии использования легко находятся, как и то, что вы сможете легко договориться с другими, какие сценарии использования следует отнести к базовым, а какие к обычным. Базовый сценарий использования почти всегда будет своего рода абстракцией других сценариев использования; возможно, для него даже потребуется новый термин или новое имя, по которому его можно будет отличить от других. Даже если вы получили спецификацию требований, в которой отсутствуют многие сценарии использования, в этом несовершенном документе будут содержаться базовые сценарии, потому что в них проявляется сущность бизнеса. Кроме того, хотя вы не должны проектировать под конкретные требования, это не означает, что на требования можно не обращать внимания. Вся суть анализа требований — выявление базовых сценариев использования (и областей нестабильности). Ваша задача как

архитектора (вместе с владельцем требований) — выявление базовых сценариев использования, часто с применением некоторого итеративного процесса.

Задача архитектора

Ваша задача как архитектора — выявить наименьший набор компонентов, которые можно объединить для выполнения всех базовых сценариев использования. Так как все остальные сценарии являются обычными разновидностями базовых сценариев использования, обычные сценарии использования просто представляют другие взаимодействия между компонентами, а не другую декомпозицию. Теперь при изменении требований ваше решение не изменится.

> **ПРИМЕЧАНИЕ** Это наблюдение относится к декомпозиции на компоненты, а не к реализации кода внутри компонентов. Например, при использовании Метода код интеграции компонентов находится в основном в Менеджерах. Скорее всего, код интеграции изменится в ответ на изменение требований. Такие изменения не являются архитектурными изменениями, это изменение реализации. Более того, степень подобных изменений реализации, обусловленных изменениями в требованиях, не зависит от качества проработки решения.

Я называю такой подход *компоновочным проектированием*. Компоновочное проектирование не стремится удовлетворить какой-то конкретный сценарий использования.

Вы не ориентируетесь на какой-либо конкретный сценарий не только потому, что полученные вами сценарии использования могут быть неполными и в них могут содержаться многочисленные ошибки и противоречия, а потому, что они изменятся. Даже если существующий сценарий использования не изменится, со временем у вас появятся новые сценарии использования, а другие будут исключены.

Простой пример — структура человеческого тела. Вид *Homo sapiens* появился на равнинах Африки более 200 000 лет назад, когда требования окружающей среды совершенно не включали квалификацию архитектора программных систем. Как же можно соответствовать требованиям архитектора программных систем в наши дни, находясь в теле охотника-собирателя? Дело в том, что хотя вы используете те же компоненты, что и доисторический человек, эти компоненты интегрируются по-другому. Тем не менее базовый сценарий использования — *выживание* — не изменился со временем.

Проверка архитектуры

Так как конечной целью любой системы является удовлетворение требований, компоновочное проектирование открывает возможность *проверки проектиро-*

вочного решения. После того как вы сконструировали взаимодействие между своими сервисами для каждого базового сценария использования, то произвели действительное решение. Нет необходимости знать неизвестное или составлять прогнозы на будущее. Ваше проектирование теперь может справиться с любым сценарием использования, потому что все сценарии использования проявляются только в разных взаимодействиях между одними структурными элементами. Не надейтесь, что в один прекрасный день вы получите сказочный проект, в котором все требования будут полностью сформулированы и правильно документированы. Нет смысла терять много времени на ранней стадии проекта, пытаясь зафиксировать все требования в мельчайших подробностях. Действительные системы можно легко проектировать даже с крайне несовершенными требованиями.

Акт проверки проектировочного решения может сводиться к построению простых диаграмм, демонстрирующих взаимодействие между компонентами архитектуры, поддерживающими сценарии использования. Диаграмма на рис. 4.1 в терминологии Метода называется *диаграммой цепочки вызовов*.

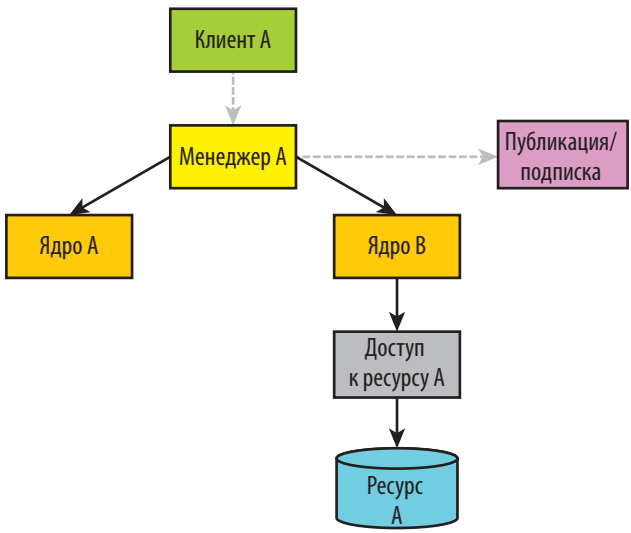


Рис. 4.1. Простая цепочка вызовов, демонстрирующая поддержку базовых сценариев использования

Цепочка вызовов демонстрирует взаимодействие между компонентами, необходимое для удовлетворения конкретного сценария использования. Вы можете буквально наложить цепочку вызовов на многоуровневую диаграмму архитектуры. Компоненты на диаграмме соединяются стрелками, обозначающими направление и тип вызовов между компонентами: сплошные черные стрелки для синхронных вызовов (запрос/ответ), пунктирная серая стрелка для вызо-

ва в очереди. Диаграммы цепочки вызовов являются специализациями графа зависимостей и как таковые весьма полезны при проектировании плана проекта (о котором речь пойдет в части II книги).

Диаграммы цепочек вызовов — простое и быстрое средство анализа сценария использования, которое демонстрирует, как этот сценарий поддерживается проектировочным решением. К недостаткам диаграмм цепочек вызовов можно отнести то, что в них не существует концепции порядка вызовов, нет способа отражения продолжительности вызовов и они становятся слишком запутанными при нескольких вызовах компонентов одного типа несколькими разными сторонами. Во многих случаях взаимодействие между компонентами может быть достаточно простым, так что вам не нужно показывать порядок, продолжительность или наличие множественных вызовов. В таких ситуациях может оказаться, что диаграммы цепочек вызовов достаточно хороши для целей проверки архитектуры. Кроме того, цепочки вызовов часто получаются более понятными для нетехнических аудиторий.

Диаграмма последовательности в терминологии Метода похожа на диаграмму последовательности UML[1]. Тем не менее она включает ряд изменений в обозначениях, обеспечивающих единство смыслов между разными типами диаграмм. «Линиям жизни» назначаются цвета в соответствии с архитектурными уровнями, а типы стрелок выбираются так же, как на диаграммах цепочки вызовов. На рис. 4.2 изображена диаграмма последовательности, эквивалентная рис. 4.1.

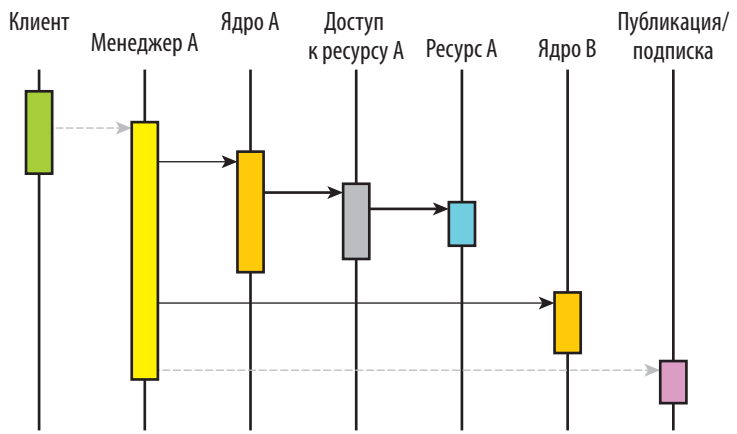


Рис. 4.2. Демонстрация поддержки базового сценария использования на диаграмме последовательности

[1] https://ru.wikipedia.org/wiki/Диаграмма_последовательности

На диаграмме последовательности каждый участвующий компонент в сценарии использования помечен вертикальной полосой, представляющей его линию жизни. Вертикальные полосы соответствуют некоторой работе или активности, выполняемой компонентом. Время течет сверху вниз на диаграмме, а длина полос обозначает относительную продолжительность использования компонента. Один компонент может многократно участвовать в одном сценарии использования, а разные экземпляры одного компонента даже могут иметь разные линии жизни. Горизонтальные стрелки обозначают вызовы между компонентами (сплошные черные — для синхронных, пунктирные серые — для очередей).

Построение диаграмм последовательности занимает больше времени из-за дополнительного уровня детализации, но часто такие диаграммы идеально подходят для демонстрации сложных сценариев использования, особенно для технической аудитории. Кроме того, диаграммы последовательности чрезвычайно полезны при последующем подробном проектировании: они помогают определять интерфейсы, методы и даже параметры. Если вы собираетесь строить их для подробного проектировочного решения, с таким же успехом вы можете построить их сначала для проверки решения, хотя и с меньшей детализацией (например, временно опустить операции и сообщения).

Наименьший набор

Помните: ваша задача как архитектора — не просто найти набор компонентов, которые можно объединить для удовлетворения всех базовых сценариев использования, а найти *наименьший* набор. Почему наименьший? И что в данном случае вообще понимать под наименьшим?

В общем случае вы должны произвести архитектуру, которая сводит к минимуму (а не к максимуму) объем работы по подробному проектированию и реализации. В том, что касается архитектуры, «лучше меньше, да лучше». С учетом сказанного для количества компонентов в архитектуре существуют естественные ограничения. Например, представьте, что вам вручена спецификация требований с 300 сценариями использования. С одной стороны, однокомпонентная архитектура, удовлетворяющая эти требования, содержит безусловно минимальное количество компонентов, но такой монолит станет ужасным проектировочным решением из-за его внутренней сложности (за подробным обсуждением влияния размера сервиса на затраты обращайтесь к приложению Б). С другой стороны, если вы создали архитектуру из 300 компонентов, каждый из которых соответствует одному сценарию использования, это тоже не может считаться признаком хорошего проектирования из-за слишком высокой стоимости интеграции. Достаточно хорошее количество должно лежать где-то в диапазоне от 1 до 300.

Несмотря на неопределенность оценки, порядковые оценки могут быть очень полезными. Например, в системе с 300 сценариями использования каким дол-

жен быть порядок величины числа компонентов для действительного решения? 1, 10, 100 или 1000 компонентов? Независимо от специфики системы, вы интуитивно знаете, что 1, 100 и 1000 — неправильные ответы, остается порядок 10. Наименьший набор сервисов, необходимых для типичной программной системы, имеет порядок величины 10 (то есть наборы из 12 и 20 компонентов имеют порядок 10). Этот конкретный порядок величины является почти универсальной концепцией проектирования. Сколько внутренних компонентов содержит ваше тело по порядку величины? Ваш автомобиль? Ваш ноутбук? Во всех перечисленных случаях ответ остается одним и тем же — 10 — по комбинаторным соображениям. Если система поддерживает требуемое поведение объединением 10 компонентов или около того, то это делает возможным невероятное число комбинаций даже без повторения задействованных компонентов или частичных наборов. В результате даже небольшое количество действительных внутренних компонентов позволит поддерживать астрономическое число возможных сценариев использования.

Возвращаясь к хорошо спроектированным программным системам: компоненты системы инкапсулируют области нестабильности. При использовании Метода даже в большой системе обычно присутствуют от двух до пяти *Менеджеров*, от двух до трех *Ядер*, от трех до восьми *Ресурсов* или компонентов *Доступ к ресурсу* и около пяти-шести *Вспомогательных средств*. Общее количество структурных элементов не должно превышать десятка, максимум двух. При большем количестве вам придется разделить систему на логически связанные подмножества (подсистемы) более удобного размера. Когда вы уже не можете предложить меньший набор структурных элементов, значит, вы нашли свое лучшее решение. Неважно, что более опытный архитектор сможет предложить еще меньший набор компонентов, потому что другой архитектор не проектирует вашу систему. Во всех работах по проектированию всегда имеется точка падения эффективности, и ваш минимальный набор находится именно в этой точке.

> **ПРИМЕЧАНИЕ** Архитектуры невозможно проверять на одном компоненте или сотнях компонентов. Один большой компонент делает все по определению, решение с одним компонентом на каждый сценарий использования тоже поддерживает все сценарии использования.

Продолжительность работы по проектированию

Вы можете провести недели и даже месяцы за попытками идентифицировать базовые сценарии использования и области нестабильности. Тем не менее эта работа не относится к проектированию — это фаза сбора и анализа требований, которая может занимать очень много времени. После того как вы определились с базовыми сценариями использования и областями нестабильности, сколько времени вам потребуется для построения действительного проектировочного

решения с применением Метода? Здесь также можно мыслить порядками: это будут часы? Дни? Недели? Месяцы? Годы? Вероятно, большинство читателей этой книги выберет день или месяц, а с практикой можно сократить время до нескольких часов. Проектирование занимает не так много времени, если вы хорошо понимаете, что делаете.

Никаких функций нет

Объединяя наблюдения этой главы с выводами двух предыдущих глав, мы приходим к фундаментальному правилу проектирования систем:

Функции всегда и везде являются аспектами интеграции, а не реализации.

Это универсальное правило проектирования, которое управляет проектированием и реализацией всех систем. Как упоминалось в главе 2, определение «универсальный» по своей природе включает программные системы.

Процесс сборки автомобилей хорошо демонстрирует это правило. Ваш автомобиль имеет критическую функцию: он должен транспортировать вас из точки А в точку Б. Если бы вы наблюдали за тем, как производятся машины, как бы увидели эту функцию? Эта функция появится после того, как вы интегрируете раму с блоком двигателя, коробкой передач, сиденьями, приборной панелью, водителем, дорогой, страховкой и топливом. Интеграция всего сказанного дает функцию.

В этом правиле еще больше впечатляет то, что оно имеет фрактальную природу. Например, я создаю эту книгу на ноутбуке, который предоставляет в мое распоряжение очень важную функцию: работу с текстом. Но есть ли в архитектуре ноутбука блок с надписью «Работа с текстом»? Чтобы предоставить функцию работы с текстом, ноутбук интегрирует клавиатуру, экран, жесткий диск, шину, процессор и память. Каждый из этих компонентов также предоставляет свою функцию: процессор — обработку чисел, жесткий диск — хранение данных. А если присмотреться к функции хранения данных, существует ли в архитектуре диска один блок с надписью «Хранение данных»? Жесткий диск предоставляет функцию хранения данных в результате интеграции внутренних компонентов: памяти, внутренней шины данных, носителей информации, кабелей, портов, регуляторов питания и маленьких винтов, которые скрепляют все компоненты. Сами по себе винты предоставляют очень важную функцию: крепление. Но как винт предоставляет эту функцию? Это делается посредством интеграции головки винта, резьбы и стержня. Интеграция всех этих аспектов предоставляет функцию крепления. Разложение можно продолжать до уровня кварков, и вы нигде так и не обнаружите функции.

Перечитайте правило проектирования, которое вы только что узнали. Если вам по-прежнему трудно его принять, значит, вы подключены к Матрице, ко-

торая приказывает вам писать код, реализующий функции. Это противоречит самому устройству Вселенной. Функции нет.

Обработка изменений

Ваша программная система должна реагировать на изменения в требованиях. Многие программные системы реализуются с применением функциональной декомпозиции, которая доводит до максимума последствия изменений. Если проектировочное решение основано на функциях, то изменения по определению никогда не проявляются в одном месте. Вместо этого они распределяются по разным компонентам и аспектам системы. С функциональной декомпозицией изменения приводят к дорогостоящим и болезненным изменениям, поэтому люди стараются избегать изменений, всячески сопротивляясь им. Они добавят запрос на изменение в следующую версию, которая должна выйти через полгода, потому что будущие неприятности лучше настоящих. Они даже пытаются немедленно сопротивляться изменениям, объясняя заказчику, что запрашиваемое изменение — неудачная идея.

К сожалению, сопротивление изменениям равносильно убийству системы. Системы, которые используются заказчиками, живут, а неиспользуемые системы мертвы (даже если за них продолжают платить). Когда разработчики сообщают заказчикам, что изменение станет частью будущей версии, что должен делать заказчик ближайшие полгода, пока разработчики трудятся над запрашиваемым изменением? Заказчикам не нужна эта функция через полгода — она нужна им сейчас. Соответственно, заказчикам придется искать обходной путь: использовать унаследованную систему, или какой-нибудь внешний носитель, или продукт конкурента. Так как сопротивление изменениям отпугивает заказчиков от системы, сопротивление изменениям равносильно ее убийству. Быстрая реакция — часть реакции на изменения, даже если этот аспект нигде не был явно сформулирован.

Ограничение изменений

Чтобы решить проблему изменений, вы не должны бороться с ними, откладывать их на будущее или отвергать. Суть в том, чтобы ограничить их последствия. Рассмотрим систему, спроектированную с использованием декомпозиции на основе нестабильности и структурных рекомендаций из главы 3. Изменение требования в действительности является изменением в требуемом поведении системы, а именно изменением в сценарии использования. В Методе некий *Менеджер* реализует процесс выполнения сценария использования. Изменения могут оказать значительное влияние на *Менеджера*. Возможно, вам даже придется уничтожить всю реализацию *Менеджера* и создать на ее месте новую версию. Тем не менее изменения в требуемом поведении не отразятся на компонентах, интегрируемых *Менеджером*.

Вспомните, о чем говорилось в главе 3: *Менеджеры* должны быть почти расходными. Это позволяет вам поглотить затраты на внесение изменений, сдержать их. Более того, основные усилия в работе над любой системой обычно направляются на сервисы, используемые *Менеджером*:

- Реализация *Ядер* обходится дорого. Каждое *Ядро* представляет бизнес-активности, критичные для процессов системы, и инкапсулирует сопутствующую нестабильность и сложность.
- Реализация *Доступа к ресурсу* нетривиальна, и дело даже не только в затратах на написание кода. Выявление атомарных бизнес-команд, перевод их в методологии обращения к *Ресурсу* и оформление в виде ресурсонезависимого интерфейса также требует значительных усилий.
- Проектирование и реализация *Ресурсов*, которые были бы масштабируемыми, надежными, высокопроизводительными и обладали высокой степенью повторного использования, требует чрезвычайно высоких затрат времени и усилий. К числу таких задач может относиться проектирование контрактов данных, схем, политик обращения к кэшу, разбиения, управления подключением, тайм-аутов, управления блокировками, индексирования, нормализации, форматов сообщений, транзакций, ошибок доставки, подозрительных сообщений и многого другого.
- Реализация *Вспомогательных средств* всегда требует высочайшей квалификации, а результат должен быть абсолютно надежным. *Вспомогательные средства* — «хребет» вашей системы. Безопасность мирового уровня, диаграмма, качественное ведение журнала, обработка сообщений, инструментальные средства управления и размещение не реализуются сами по себе.
- Проектирование выдающегося пользовательского интерфейса и удобного, пригодного для повторного использования API для *Клиентов* требует значительных затрат времени и усилий. *Клиенты* также должны взаимодействовать и интегрироваться с *Менеджерами*.

Когда с *Менеджером* происходят изменения, вам приходится перерабатывать и использовать повторно все усилия, затраченные на *Клиентов*, *Ядра*, *Доступ к ресурсам*, *Ресурсы* и *Вспомогательные средства*. Заново интегрируя эти сервисы в *Менеджере*, вы ограничиваете последствия изменений и можете быстро и эффективно реагировать на них. Разве не в этом заключается суть быстрой реакции?

5 Пример проектирования системы

В предыдущих трех главах были представлены универсальные принципы проектирования систем. Тем не менее люди обычно предпочитают учиться на конкретном примере. По этой причине в этой главе будет продемонстрировано применение концепций из предыдущих глав на подробном практическом примере. Этот пример описывает проектирование новой системы TradeMe, которая разрабатывается в качестве замены для старой системы. Прототипом примера стала реальная система, которую компания IDesign спроектировала для одного из своих заказчиков, хотя и с удалением и маскировкой конкретных подробностей. Суть системы осталась неизменной, от обоснования проекта до декомпозиции: я не стал скрывать некоторые проблемы и не пытался приукрасить ситуацию. Как упоминалось в главе 1, проектирование не должно занимать слишком много времени. В данном случае оно было завершено менее чем за неделю группой из двух человек: опытного архитектора IDesign и стажера.

Целью примера является демонстрация мыслительных процессов и умозаключений, используемых для построения проектировочного решения. Обычно изучать такие вещи самостоятельно непросто, но они становятся более понятными во время наблюдения за кем-то одновременно с наблюдением относительно происходящего. Глава начинается с общего обзора заказчика и системы, после чего требования к ней представляются в форме нескольких сценариев использования. Для выявления областей нестабильности и архитектуры используется структура Метода.

> **ВНИМАНИЕ** Не пытайтесь применять этот пример догматически, как шаблон для самостоятельной работы. Все системы отличаются друг от друга, для каждой системы действуют свои ограничения, разные определяющие факторы и потенциальные компромиссы. Ваша полезность как архитектора проявляется тогда, когда вы создаете правильное проектировочное решение для имеющейся системы. Это требует практики и критического мышления. В этой главе вам стоит сосредоточиться на обоснованиях для принимаемых решений и использовать этот пример как основу для дальнейшей самостоятельной работы, как обсуждалось в главе 2.

Обзор системы

TradeMe — система, связывающая мастеров с подрядчиками и проектами. Мастерами могут быть сантехники, электрики, плотники, сварщики, маляры, садовники, установщики солнечных панелей и т. д. Все они работают независимо и являются индивидуальными предпринимателями. Каждый мастер характеризуется уровнем квалификации, а некоторые (например, электрики) были сертифицированы регламентирующими структурами для выполнения некоторых операций. Ставка оплаты мастера зависит от различных факторов: направления (сварщики получают больше, чем плотники), квалификации, стажа, типа проекта, местонахождения и даже погоды. Также учитываются и другие факторы, включая соответствие действующему законодательству (минимальная ставка или налоги), премии за риск (например, работа на стенах небоскребов или высокое напряжение), наличие сертификатов для выполнения некоторых задач (например, сварка несущих балок или развязок электросетей), требования к составлению отчетности и т. д.

У подрядчиков возникает ситуативная необходимость в услугах мастеров, от одного дня до нескольких недель. У них часто имеется базовая команда специалистов, которые используются для штатной работы, а TradeMe задействуется для специализированной работы. В одном проекте разные мастера нужны на разные периоды времени (один на день, другой на неделю) в разное время. Мастера могут приходить и уходить в рамках одного проекта.

Система TradeMe позволяет мастерам регистрироваться, указывать свою квалификацию, общую географическую зону доступности и желательный размер оплаты. Она также позволяет подрядчикам регистрироваться, публиковать информацию о вакансиях, местонахождении проектов, желательном размере оплаты, продолжительности работы и других атрибутах проекта. Подрядчики даже могут запрашивать (но не настаивать) конкретных мастеров, с которыми они бы предпочли работать.

Помимо уже перечисленных факторов, величина оплаты, которую готов платить подрядчик, зависит от спроса и предложения. Если проект простаивает, подрядчик повышает цену. Если мастер простаивает, то он понижает запрашиваемую цену. Аналогичные соображения применяются к продолжительности или запрашиваемым обязательствам. Идеальный проект для мастера часто характеризуется высокой оплатой и малой продолжительностью. После того как мастер согласился на участие в проекте, он должен работать в течение зафиксированного промежутка времени. Подрядчики могут предлагать более высокую оплату с большей продолжительностью работы. В общем случае система позволяет рыночным факторам отрегулировать величину оплаты и найти баланс запроса и предложения.

Целью проектов является строительство зданий. Система также может оказаться полезной в условиях формирующихся рынков (например, недавно обнаруженных нефтяных месторождений).

TradeMe позволяет мастерам и подрядчикам найти друг друга. Система обрабатывает запросы и обеспечивает доставку необходимых мастеров к месту работы. Она также отслеживает рабочие часы и ставки, а еще предоставляет отчетность для контролирующих органов, избавляя как подрядчиков, так и мастеров от хлопот с самостоятельным решением этих задач.

Система изолирует мастеров от подрядчиков. Она получает средства от подрядчиков и оплачивает работу мастеров. Подрядчики не могут обойти систему и нанять мастеров напрямую, потому что мастерам предоставлены особые привилегии в системе. Система TradeMe пытается найти наилучшую оплату для мастеров и наилучшую доступность для подрядчиков. Она зарабатывает на небольшом расхождении между запрашиваемой и заявленной оплатой. Другим источником дохода являются членские взносы, которые оплачиваются как мастерами, так и подрядчиками. Взносы собираются ежегодно, но частота сбора может измениться. Как мастера, так и подрядчики называются *участниками* в системе.

В настоящее время большинство назначений обрабатывается девятью контактными центрами. Каждый контактный центр обслуживает конкретный географический регион с определенными законодательными нормами, строительными нормами, стандартами и трудовым законодательством. Контактные центры укомплектованы штатными консультантами, которые называются *операторами*. В настоящее время операторы оптимизируют планирование по всем проектам и доступным мастерам на основании своего опыта. Некоторые контактные центры работают как самостоятельные организации, другие находятся под управлением одной организации.

Также существует как минимум одно конкурентное приложение, ориентированное на поиск самых дешевых мастеров; некоторые подрядчики предпочитают такую систему. Подрядчики, выбирающие мастеров по критерию цены (вместо доступности), могут стать усиливающейся тенденцией.

Унаследованная система

У унаследованной системы, развернутой в европейских контактных центрах, имеются постоянные пользователи. Они работают с двухуровневым настольным приложением, подключенным к базе данных. К ним поступают обращения как от мастеров, так и от подрядчиков. Операторы вводят информацию и даже проводят поиск предложений в реальном времени. Для управления включением в систему используются рудиментарные веб-порталы, которые обходят старую систему и работают с базой данных напрямую. Различные подсистемы изолированы и работают крайне неэффективно, требуя значительного участия со стороны человека почти на каждом шаге. Пользователь должен использовать до пяти разных приложений для решения своих задач. Эти приложения независимы друг от друга, и вся интеграция осуществляется

вручную. Клиентские приложения плотно набиты бизнес-логикой, а отсутствие разделения между пользовательским интерфейсом и бизнес-логикой не позволяет обновить приложения до современных стандартов пользовательского интерфейса.

У каждой подсистемы даже имеется собственный репозиторий, и пользователям приходится согласовывать хранящиеся в них данные, чтобы извлечь из них хоть какой-нибудь смысл. Этот процесс сопряжен с высоким риском ошибок, требует дорогостоящего обучения и значительного времени вхождения.

Старая система уязвима, а ее бессистемный подход к безопасности открывает ее для многих разных векторов атаки. Старая система не проектировалась с учетом безопасности. Если на то пошло, она вообще не проектировалась, а выросла естественным образом.

Старая система попросту была не способна поддерживать некоторые новые и требуемые возможности:

- Поддержка мобильных устройств.
- Более высокая степень автоматизации процесса.
- Возможность подключения к другим системам.
- Миграция в облачную платформу.
- Выявление попыток мошенничества.
- Анализ качества работы, включая учет показателей безопасности в оплате и требований к квалификации.
- Выход на новые рынки (например, развертывание на верфях малотоннажного судостроения).

Как бизнес, так и пользователей раздражает неспособность старой системы идти в ногу со временем, поэтому поток запросов на реализацию функций никогда не иссякает. Одна из таких функций — повышение квалификации — оказалась совершенно обязательной, поэтому она была построена поверх старой системы. Старая система назначает мастеров на курсы повышения квалификации и сдачу обязательных экзаменов, а также отслеживает процесс обучения мастеров. Хотя внешние образовательные центры предоставляли услуги обучения и регистрации сертификатов, в старой системе пользователям приходилось связываться с ними вручную. Хотя эта функция не связана напрямую с базовыми аспектами системы, мастера очень сильно заинтересованы в ней — как и бизнес, потому что функция сертификации поможет предотвратить уход мастеров к конкурентам.

У старой системы также возникали трудности с выполнением новых законодательных норм в разных локальных контекстах. Реализация любых изменений была крайне затруднена, а система сильно привязана к текущему бизнес-

контексту. Так как компания не может позволить себе поддержку уникальной версии системы для каждого локального контекста, появился стимул к «усечению» системы до некоего общего минимума по разным локальным контекстам. Это привело к повышению нагрузки на пользователей в отношении их потоков операций, выполняемых вручную; это привело к повышению времени обучения и затрат, а также к потере некоторых бизнес-возможностей.

Во всех филиалах системы работают около 220 операторов. Ни с масштабируемостью, ни с пропускной способностью особых проблем нет. Но скорость реакции оставляет желать лучшего, хотя это может быть побочным эффектом использования старой системы.

Новая система

Руководство компании понимает недостатки плохо спроектированной старой системы и желает, чтобы новая система была спроектирована правильно. Новая система должна автоматизировать как можно большую часть работы. В идеале в компании должен быть один небольшой контактный центр, который используется в качестве резерва для автоматического процесса. Этот контактный центр должен использовать одну систему для всех локальных контекстов. Хотя система была развернута в Европе, поступали заявки на ее внедрение в Великобритании[1] и даже в Канаде (то есть за пределами Евросоюза). Другой определяющий фактор для вложений в новую систему заключался в том, что конкуренты разработали более гибкие, более эффективные и удобные для пользователя системы.

Хотя подрядчики могли укомплектовать персонал проекта из нескольких источников мастеров, включая продукты конкурентов, интеграция с продуктами конкурентов и оптимизация проектов вообще выходят за рамки системы: компания не специализируется в отрасли оптимизации или интеграции. Расширение рынка для включения в него других сфер услуг (например, IT или ухода за детьми) также не рассматривается. Добавление этих аспектов переопределило бы природу бизнеса, а компания специализируется на подборе мастеров для строительных проектов, а не на подборе персонала вообще.

Компания

Компания рассматривает себя как посредника при поиске персонала, а не как разработчика. Программирование не является ее основной деятельностью. В прошлом у компании не было четкого понимания того, что требуется для

[1] Хотя система проектировалась до Брексита (выхода Великобритании из Евросоюза), Брексит служит классическим примером серьезных изменений, которые было сложно предвидеть на момент проектирования. При этом новая система адаптировалась к этому изменению без малейших проблем.

разработки качественного программного продукта. Процессу или практике разработки не уделялось достаточного внимания. Все прошлые попытки компании разработать систему замены завершились неудачей. При этом компания располагает финансовыми ресурсами — старое приложение было чрезвычайно прибыльным. Горькие уроки прошлого убедили руководство сменить подход и более основательно подойти к разработке.

Сценарии использования

Ни для старой, ни для новой системы не было готовой документации со списком требований. Заказчик предоставил только схемы (рис. 5.1–5.8), где были даны некоторые сценарии использования. Было неясно, являются ли эти сценарии базовыми; они просто описывали требуемое поведение системы. В значительной степени в этих сценариях использования отражалось то, как должна была работать старая система. Так как группу проектирования интересовали базовые сценарии использования, она игнорировала такие низкоуровневые сценарии, как ввод финансовых данных, получение оплаты от подрядчиков и распределение оплаты среди мастеров. Некоторые сценарии использования (например, повышение квалификации) даже не определялись. Более того, в системе явно оставалось место для дополнительных сценариев использования, которые бы дополняли сценарии, предоставленные компанией.

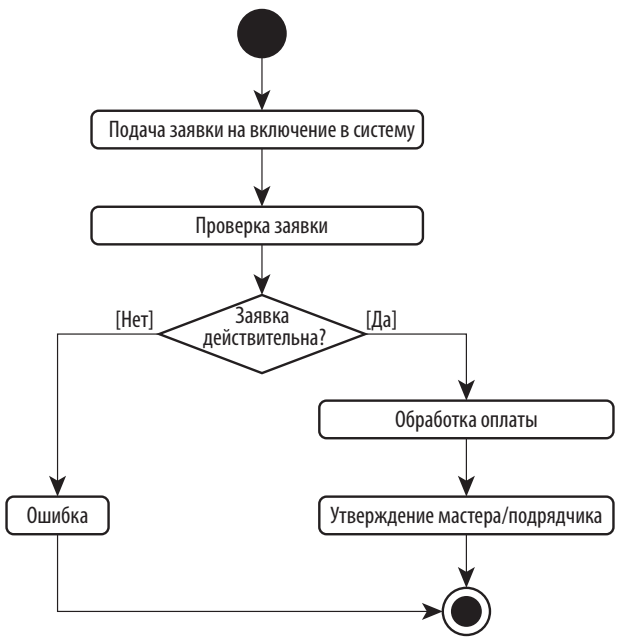


Рис. 5.1. Сценарий использования для добавления подрядчика или мастера

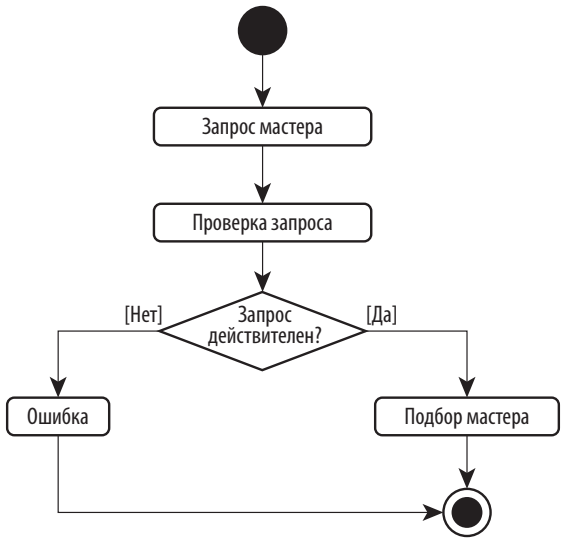


Рис. 5.2. Сценарий использования для запроса мастера или подрядчика

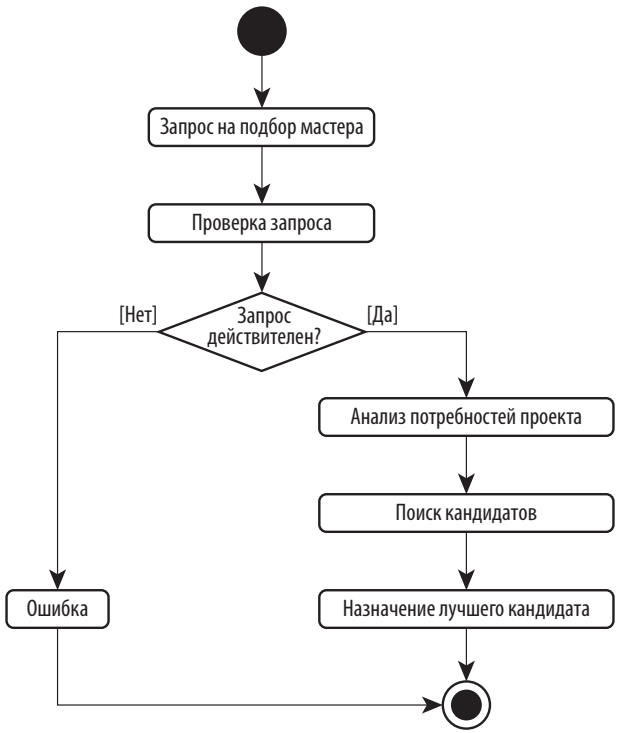


Рис. 5.3. Сценарий использования для подбора мастера

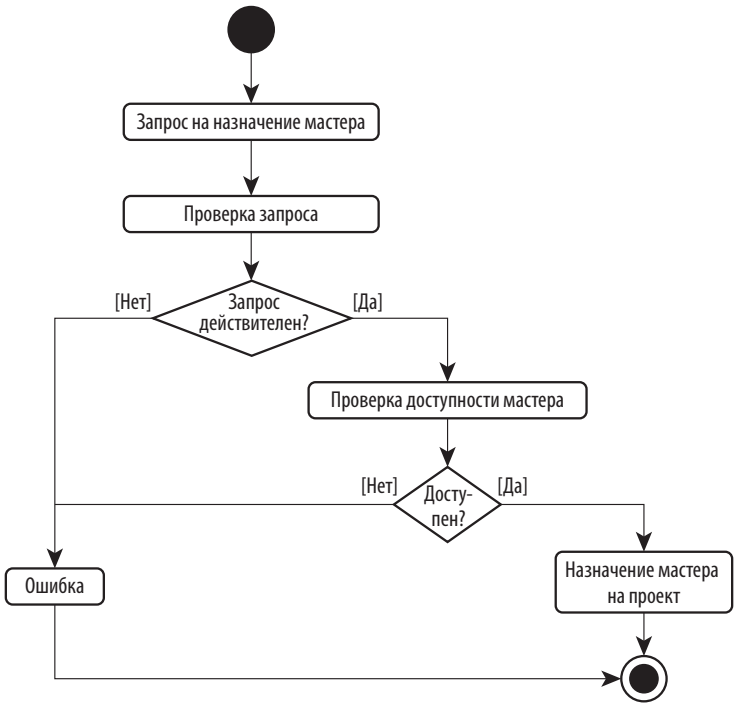


Рис. 5.4. Сценарий использования для назначения мастера

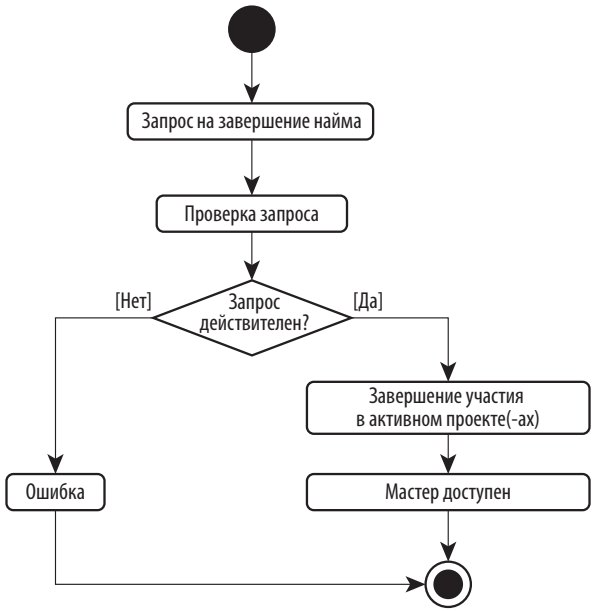


Рис. 5.5. Сценарий использования для завершения найма мастера

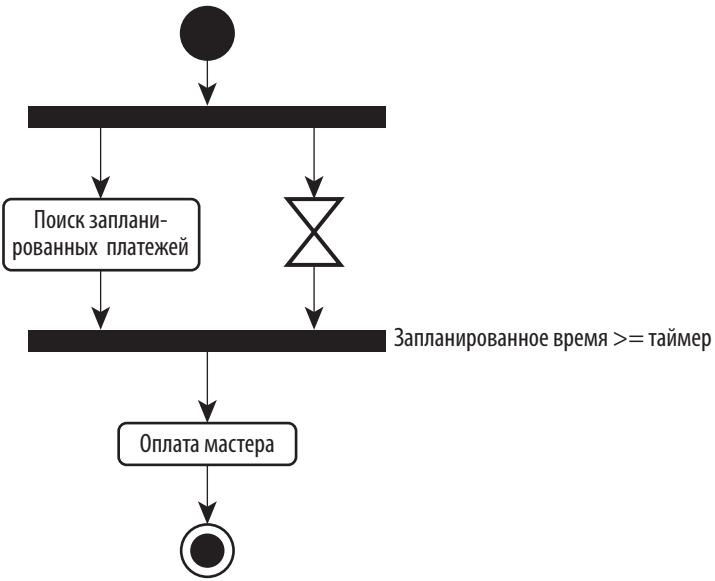


Рис. 5.6. Сценарий использования для оплаты мастера

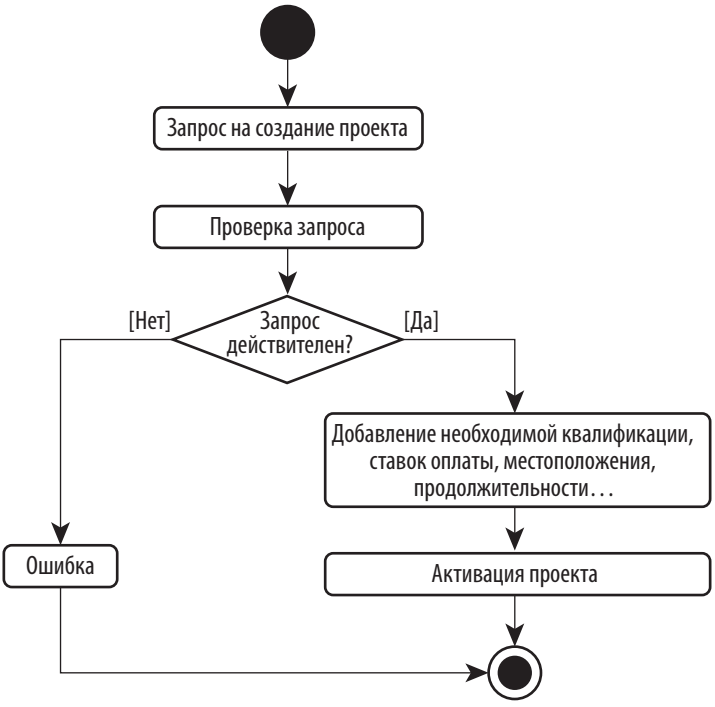


Рис. 5.7. Сценарий использования для создания проекта

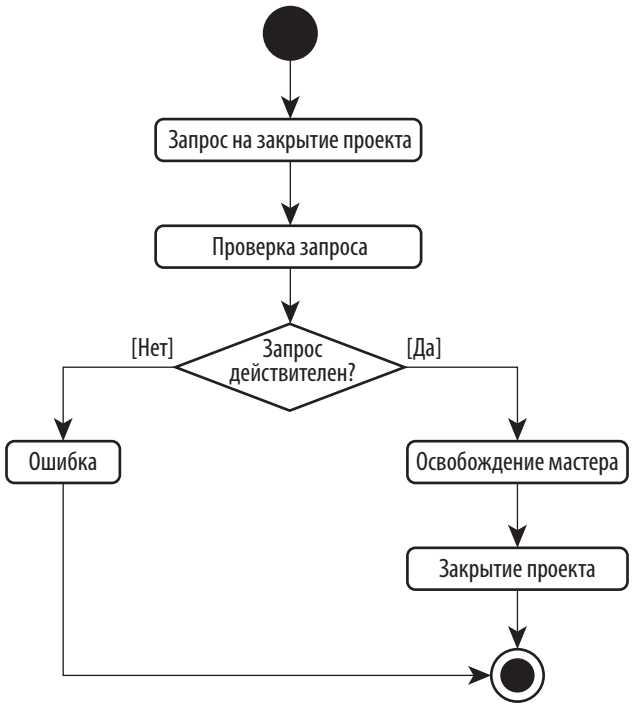


Рис. 5.8. Сценарий использования для закрытия проекта

ПРИМЕЧАНИЕ Как упоминалось в главе 4, полученный набор требований редко бывает идеальным (система TradeMe не была исключением). Даже сколько-нибудь приличный список сценариев встречается редко. Одна из главных целей этой главы — показать, как построить эффективную архитектуру даже при такой неопределенности.

Базовый сценарий использования

Многие сценарии использования, предоставленные компанией, были похожи не на базовые сценарии, а скорее на простые описания функциональности. Вспомните, что базовый сценарий использования представляет сущность бизнеса. Сущность системы не заключается в добавлении мастеров или подрядчиков, создании проектов или оплате мастеров. Все эти задачи могут решаться множеством разных способов; они не добавляют бизнес-ценности и не отличают систему от предложений конкурентов. Смысл существования системы выражается определением из одного предложения: «TradeMe — система, связывающая мастеров с подрядчиками и проектами». Единственным сценарием использования, хоть сколько-нибудь напоминающим это описание, был сценарий использования для подбора мастера (см. рис. 5.3).

ПРИМЕЧАНИЕ Хотя для проверки проектировочного решения достаточно только поддержки базовых сценариев использования, это не означает, что о других сценариях можно забыть, вовсе нет. Отличный способ продемонстрировать гибкость вашего решения — показать, как легко оно будет поддерживать все другие сценарии использования и все остальное, что может потребовать бизнес от вашей системы.

Упрощение сценариев использования

Заказчики редко представляют требования в полезном формате, не говоря уже о представлении, которое бы способствовало качественному проектированию. Полученные данные всегда должны подвергаться преобразованию, очистке и консолидации. На ранней стадии процесса проектирования даже можно распознать области взаимодействий, которые позднее сделают отображение областей на подсистемы или уровни намного более естественным. Например, в случае с TradeMe во всех сценариях использования наблюдались как минимум три типа ролей: пользователи, рынок и участники. Пользователями могут быть операторы по вводу данных или системные администраторы. Вероятно, только администратор сможет разорвать контракт с мастером, но на рис. 5.5 эта информация не отражена. Полезно обозначить передачу управления между ролями, организациями и другими ответственными сторонами при помощи «дорожек» на диаграммах активности. Например, на рис. 5.9 изображен альтернативный способ выражения сценария использования для завершения найма мастера, представленного на рис. 5.5.

Низкоуровневый сценарий использования преобразуется разделением диаграммы активности на области взаимодействий. Это также помогает прояснить требуемое поведение системы добавлением блоков принятий решений или полос синхронизации. Позднее в этой главе будет показано, как использовать дорожки для начала процесса проектирования и проверки результата.

Антипроектирование

В главе 2 антипроектирование упоминалось как эффективный метод демонстрации недостатков функциональной декомпозиции, основанный на попытке спроектировать наихудшую систему из всех возможных. Хотя качественное антипроектирование создает действительное проектировочное решение, потому что результат поддерживает все сценарии использования, оно не обеспечивает инкапсуляции и реализует сильное связывание. Такой подход часто кажется естественным другим людям (то есть они бы в результате проектирования создали нечто подобное). Весьма вероятно, что антипроектирование окажется некоторой разновидностью функциональной декомпозиции.

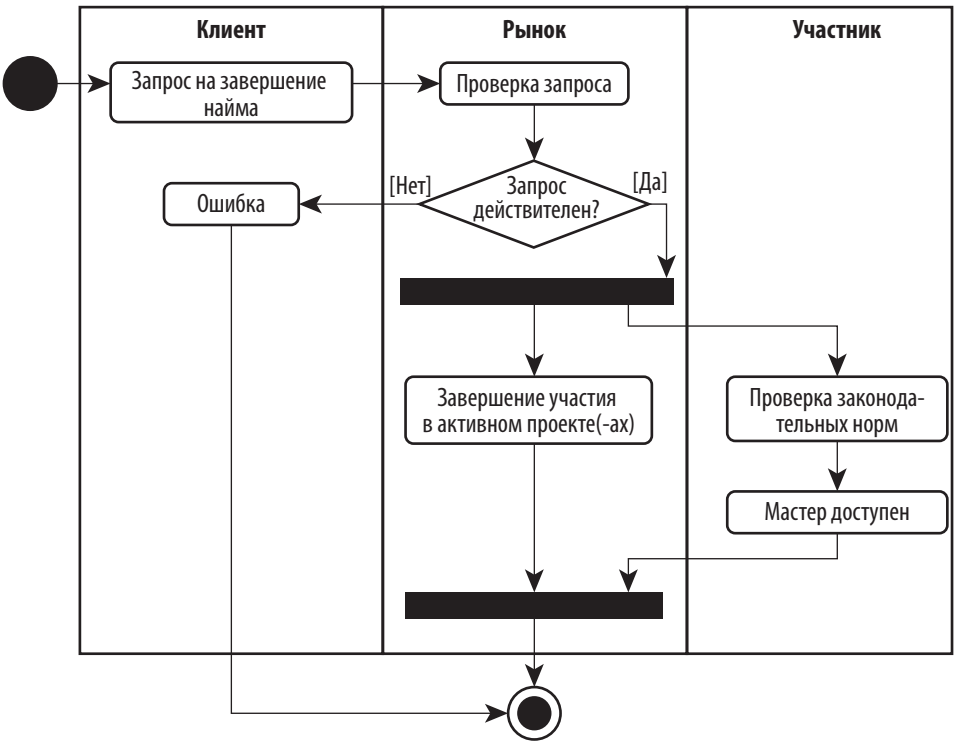


Рис. 5.9. Разделение диаграммы активности при помощи дорожек

Монолит

Простым примером антипроектирования является «всемогущий сервис» — уродливая свалка всех видов функциональности в требованиях, реализованных в одном месте. Хотя такая архитектура встречается настолько часто, что для нее даже придумано специальное название («монолит»), большинство проектировщиков на собственном горьком опыте поняли, что так лучше не делать.

Детализированные структурные элементы

На рис. 5.10 представлена другая попытка антипроектирования: огромный набор структурных элементов. Практически для каждой активности в сценариях использования в архитектуре имеется соответствующий компонент. Ни работа с базами данных, ни сама база данных не инкапсулируются. При таком количестве детализированных блоков *Клиенты* отвечают за реализацию бизнес-логики сценариев использования, как показано на рис. 5.11. Загрязнение кода *Клиента* бизнес-логикой приводит к разбуханию *Клиента*, при котором вся система мигрирует в *Клиента* (см. рис. 2.1).

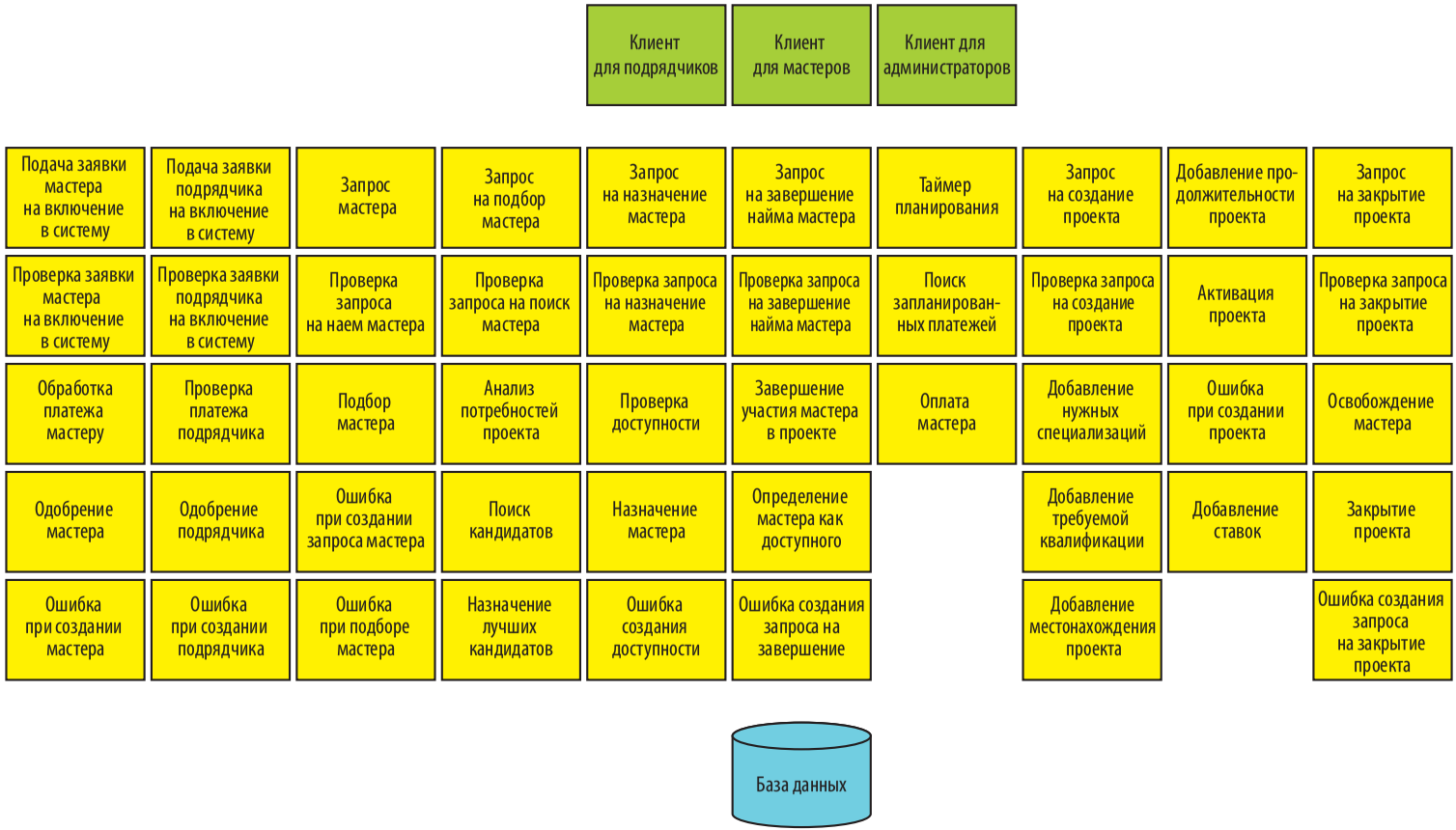


Рис. 5.10. Антипроектирование со стремительным ростом численности сервисов

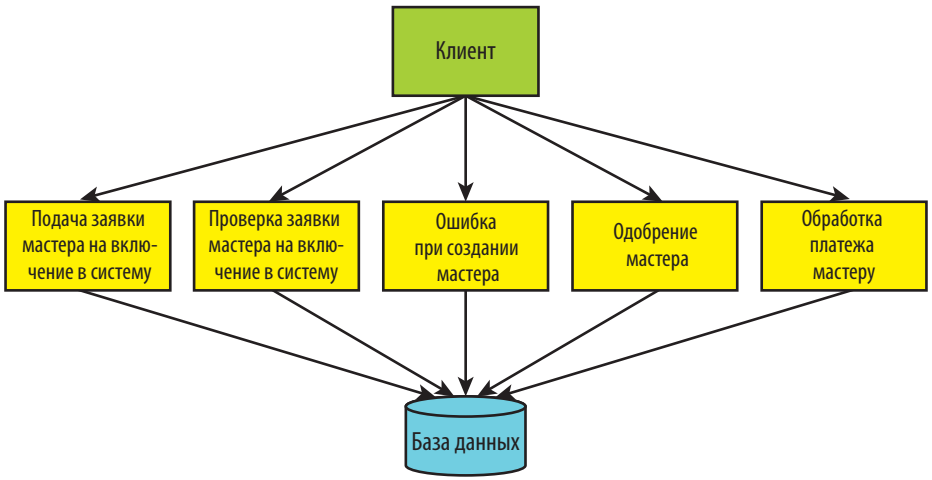


Рис. 5.11. Раздутый клиент

Также возможно решение, в котором сервисы вызывают друг друга, как показано на рис. 5.12. Однако такое сцепление высокофункциональных сервисов усиливает связи между ними, как показано на рис. 2.5. Также обратите внимание на проблемы открытой архитектуры с восходящими и горизонтальными вызовами на рис. 5.12.

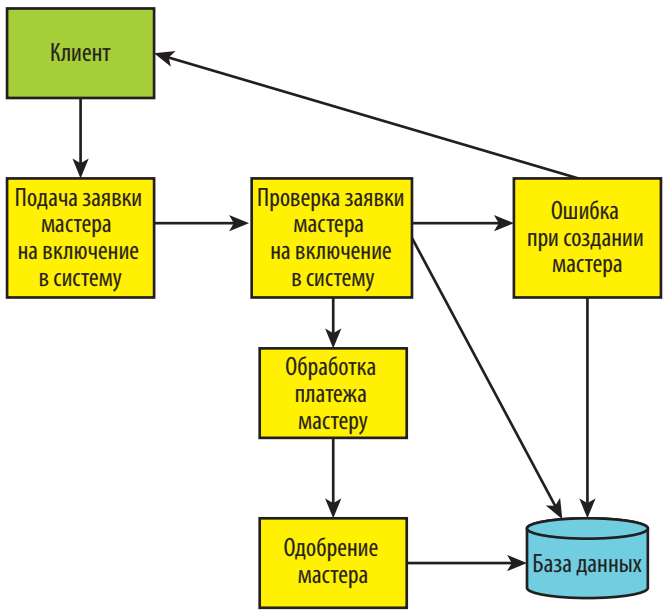


Рис. 5.12. Антипроектирование со сцеплением сервисов

Декомпозиция предметной области

Другой классический вариант антипроектирования основан на выполнении декомпозиции по границам предметных областей, как показано на рис. 5.13. В данном случае декомпозиция системы проводится по линиям предметных областей *Мастер*, *Подрядчик* и *Проект*.

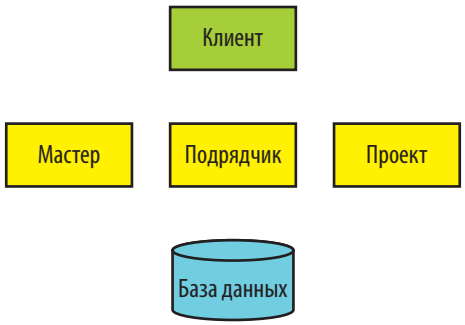


Рис. 5.13. Антипроектирование с декомпозицией предметной области

Даже в относительно простой системе (такой, как TradeMe) существуют почти неограниченные дополнительные возможности декомпозиции предметной области: *Счета*, *Администрирование*, *Аналитика*, *Одобрение*, *Сертификаты*, *Контракты*, *Валюта*, *Разногласия*, *Финансы*, *Исполнение*, *Законодательство*, *Начисление средств*, *Отчеты*, *Подбор персонала*, *Подписка* и т. д. Кто рискнет утверждать, что *Проект* — лучший кандидат для предметной области, чем *Счета*? И по какому критерию должно приниматься такое решение?

Кроме многих недостатков, рассмотренных в главе 2, с декомпозицией предметной области будет практически невозможно проверить проектировочное решение, демонстрируя поддержку сценариев использования. Например, запрос мастера появится в сервисах обеих предметных областей, *Проект* и *Мастер*. Из-за дублирования функциональностей через границы предметных областей становится не очевидно, кто, что и когда делает.

Ориентация на потребности бизнеса

Исключительно важно осознать, что архитектура не существует ради самой себя. Архитектура и система должны удовлетворять потребности бизнеса. Удовлетворение потребностей бизнеса — путеводная нить для любых усилий в области проектирования. Соответственно, вы должны позаботиться о том, чтобы архитектура была ориентирована на концепцию, которую бизнес видит в своем будущем, и на бизнес-цели. Кроме того, необходимо обеспечить двустороннюю отслеживаемость между бизнес-целями и архитектурой. Вы

должны сразу видеть, как каждая цель тем или иным способом поддерживается на уровне архитектуры и как каждый аспект архитектуры выводится из некоторых целей бизнеса. Альтернатива — бессмысленные проектировочные решения и игнорируемые потребности бизнеса.

Как упоминалось в предыдущих главах, архитектор, строящий проектировочное решение, должен прежде всего распознать области нестабильности, а затем инкапсулировать эти области в компонентах системы, оперативных концепциях и инфраструктуре. Интеграция компонентов — то, что обеспечивает требуемое поведение, а конкретный способ интеграции реализует бизнес-цели. Например, если в число ключевых целей входит расширяемость и гибкость, то интеграция компонентов по шине сообщений станет хорошим решением (подробнее об этом позднее). И наоборот, если ключевой целью является быстродействие и простота, включение шины сообщений введет слишком большую сложность.

В оставшейся части этой главы приведено подробное описание действий по преобразованию потребностей бизнеса в архитектуру TradeMe. Действия начинаются с формулировки концепции системы и бизнес-целей, которые в дальнейшем влияют на решения по проектированию.

Концепция

Редко когда все окружающие разделяют единую концепцию того, что должна делать система. У некоторых концепции нет вообще. Другие могут иметь собственную концепцию, которая служит только их узким интересам. Третьи могут неверно интерпретировать бизнес-цели. Компания, стоявшая за TradeMe, была загнана в угол множеством различных проблем, происходящих от невозможности угнаться за изменяющимся рынком. Эти проблемы были отражены в существующих системах, в структуре системы и в организации процесса разработки. Новая система должна решить все проблемы напрямую, а не по частям, потому что решения только части из них было недостаточно для успеха.

Первая потребность бизнеса — согласование общей концепции среди всех ключевых участников. Концепция должна управлять всем, от архитектуры до обязательств. Все, что вы будете делать позднее, должно служить этой концепции и оправдываться ею. Конечно, это действует в обе стороны — вот почему так желательно начать с концепции. Если что-то не служит концепции, то часто это обусловлено политическими или другими вторичными факторами. Тем самым перед вами открывается отличный способ отклонения неактуальных требований, которые не поддерживают согласованную концепцию. В случае TradeMe команда проектирования отфильтровала концепцию до одного предложения:

> *Платформа для построения приложений, обеспечивающих работу рынка TradeMe.*

Хорошая концепция компактна и явно выражена. Она должна читаться как юридическая формулировка.

Обратите внимание на то, что концепция TradeMe направлена на создание платформы для построения приложений. Такое «платформенное мышление» было ориентировано на цели разнообразия и расширяемости, столь желательные для бизнеса; вы можете применять его в проектируемых вами системах.

Бизнес-цели

После согласования концепции (и только тогда) ее можно разбить на конкретные цели. Отвергайте все цели, которые не служат концепции; включайте все цели, необходимые для поддержки концепции. Эти два типа обычно легко узнать. При перечислении целей следует руководствоваться точкой зрения бизнеса. Не позволяйте техническим специалистам или отделу маркетинга взять диалог под контроль, включать в него технологические цели или конкретные требования. Команда проектирования выделила следующие цели из общего обзора системы TradeMe:

1. Объединение репозиториев и приложений. В старой системе было слишком много неэффективных аспектов, которые требовали частого вмешательства человека для обновления и поддержания работоспособности системы.
2. Быстрая реакция на новые требования. У старой системы время реакции для новых функций было ужасным. Новая платформа должна поддерживать очень быстрые и частые обновления, часто адаптированные под конкретную квалификацию, день недели, тип проекта и любую комбинацию этих факторов. В идеале большая часть быстрой реакции должна быть автоматизирована на всех стадиях от программирования до развертывания.
3. Поддержка высокой степени адаптации для разных стран и рынков. Локализация создавала невероятные трудности из-за различий в нормах, законодательстве, культурах и языках.
4. Поддержка полной прозрачности и подотчетности перед интересами бизнеса. Обнаружение попыток мошенничества, аудиторская сквозная проверка и контроль отсутствовали в старой системе.
5. Упреждающая реакция на изменения технологии и законодательных норм. Вместо того чтобы пребывать в режиме постоянного реагирования, система должна предвидеть изменения. Компания рассчитывала, что именно за счет этого фактора TradeMe сможет одолеть своих конкурентов.
6. Хорошая интеграция с существующими системами. И хотя этот пункт отчасти связан с предыдущим, цель в данном случае заключается в обеспечении высокой степени автоматизации процессов, которые ранее выполнялись вручную и были весьма трудоемкими.

7. Оптимизация средств безопасности. Система должна быть нормально защищена, и буквально каждый ее компонент должен проектироваться с учетом безопасности. Чтобы соответствовать целям в области безопасности, группа разработки должна ввести в жизненный цикл программы такие средства, как аудит безопасности, и поддерживать их на уровне архитектуры.

> **ПРИМЕЧАНИЕ** Стоимость разработки не входила в число целей этой системы. Хотя никто не любит терять деньги, основные претензии бизнеса перечислены в приведенном списке. Компания могла позволить себе дорогостоящее решение этих проблем.

Формулировка миссии

Как ни странно, выражения концепции (того, что получит бизнес) и целей (для чего бизнесу нужна концепция) часто оказывается недостаточно. Люди обычно слишком сильно вязнут в подробностях, чтобы увидеть общую картину. Из-за этого также следует привести формулировку миссии (как вы собираетесь это сделать). Формулировка миссии TradeMe выглядит так:

> *Проектирование и построение набора программных компонентов, из которых команда разработки сможет компоновать приложения и функциональность.*

В этой формулировке разработка функций намеренно не включается в миссию. Миссия заключается не в построении функций, а в построении *компонентов*. С такой формулировкой становится оправданной декомпозиция на основе нестабильности, которая соответствует формулировке миссии, потому что она связывает все воедино:

Концепция → Цели → Формулировка миссии → Архитектура.

Такой подход фактически подталкивает бизнес к тому, чтобы предоставлять вам информацию для проектирования правильной архитектуры. Эта динамика противоположна обычной, при которой архитектор пристает к руководству с вопросами, чтобы избежать функциональной декомпозиции. Гораздо проще управлять правильностью архитектуры посредством согласования архитектуры с концепцией бизнеса, бизнес-целями и формулировкой миссии. После того как будет согласована концепция, цели и, наконец, формулировка миссии, бизнес будет на вашей стороне. Если вы хотите, чтобы люди со стороны бизнеса поддерживали ваши усилия по разработке архитектуры, вы должны продемонстрировать им, как ваша архитектура служит интересам бизнеса.

Архитектура

Недопонимание и путаница, свойственные для разработки программного обеспечения, часто ведут к конфликтам или обманутым надеждам. Маркетинг может использовать для тех же понятий другие термины, в отличие от технической стороны, или, что еще хуже, — те же термины с совершенно другим смыслом. Такие неоднозначности могут оставаться незамеченными годами. Прежде чем углубляться в проектирование системы, убедитесь в том, что все участники одинаково понимают происходящее, — для этого стоит составить краткий глоссарий терминологии предметной области.

Глоссарий TradeMe

Работу над глоссарием удобно начать с ответов на четыре классических вопроса: «кто», «что», «как» и «где». Чтобы найти ответы на эти вопросы, стоит проанализировать обзор системы, сценарии использования и примечания по собеседованиям с заказчиками, если они имеются. Для системы TradeMe ответы на четыре вопроса выглядели так:

- Кто:
  - Мастера.
  - Подрядчики.
  - Операторы TradeMe.
  - Учебные центры.
  - Фоновые процессы (например, планировщик для оплаты).
- Что:
  - Участие мастеров и подрядчиков.
  - Рынок строительных проектов.
  - Сертификаты и курсы повышения квалификации.
- Как:
  - Поиск.
  - В соответствии с законодательными нормами.
  - Обращения к ресурсам.
- Где:
  - Локальная база данных.
  - Облачные платформы.
  - Другие системы.

Вспомните, о чем говорилось в главе 3: ответы на четыре вопроса часто можно связать с уровнями, если не с компонентами самой архитектуры.

Список «что» представляет особый интерес, потому что он подсказывает возможные подсистемы (или «дорожки»). Дорожки и ответы могут использоваться для инициирования работы по декомпозиции, когда вы ищете области нестабильности. Это не препятствует созданию дополнительных подсистем и не подразумевает, что таким образом будут получены все необходимые подсистемы, — декомпозиция всегда осуществляется на основании нестабильности, а если в пункте «что» отсутствует нестабильность, он не заслуживает создания компонента в архитектуре. На этой стадии список всего лишь станет хорошей отправной точкой для рассуждений о вашем проектировочном решении.

Области нестабильности в TradeMe

Сущностью декомпозиции является выявление областей нестабильности так, как описано в предыдущих главах. В следующем списке перечислены некоторые потенциальные области нестабильности TradeMe и факторы, которые рассматривались проектировочной командой:

- *Мастер*. Можно ли считать мастеров областью нестабильности в системе? Вряд ли можно утверждать, что архитектура, пусть даже чисто функциональная, в значительной степени пострадает, если потребуется добавить новые атрибуты для мастеров. Другими словами, мастера могут изменяться, но не являются нестабильными. Это относится к любому подмножеству атрибутов мастеров (например, набору навыков). Пожалуй, мастеров нельзя считать нестабильными в изоляции. Возможно, существуют более общие нестабильности (например, управление принадлежностью к системе или законодательными нормами), ассоциированные с мастерами. Важно обсудить кандидатов на нестабильность и даже поставить их под сомнение. Если вы не можете явно сформулировать, что является областью нестабильности, почему она нестабильна и какой риск создает нестабильность с учетом ее вероятности и эффекта, значит, нужно искать дальше. Выявление мастеров как области нестабильности указывает на декомпозицию по границам предметных областей (рис. 5.12).
- *Образовательные сертификаты*. Нестабилен ли процесс сертификации? Если да, то какова истинная нестабильность с точки зрения бизнеса и системы? В данном случае нестабильность возникает в процессе сопоставления норм, управляющих необходимыми сертификатами для проектов, с мастерами, имеющими соответствующие сертификаты. Сама сертификация является атрибутом мастера. С точки зрения бизнеса управление сертификацией всегда будет вторичным фактором по отношению к базовой ценности системы — выполнению роли посредника при поиске персонала.

- *Проекты*. Заслуживает ли нестабильность проекта собственного *Менеджера*? *Менеджер проекта* подразумевает существование контекста проекта. *Менеджер рынка* более уместен, потому что какие-то активности, которыми должна управлять система, могут не требовать контекста текущего проекта для своего выполнения. Например, вы можете приказать найти подходящий вариант без указания конкретного проекта, или поиск совпадения может потребовать анализа нескольких проектов. Возможно, для того, чтобы удержать ценного мастера, вы захотите уплатить ему предварительный гонорар независимо от проекта. В определении проектов как области нестабильности проявляется декомпозиция предметной области. Базовая нестабильность — рынок, а не проекты.

Нет ничего ошибочного в том, чтобы предложить некоторые области нестабильности, а затем проанализировать полученную архитектуру. Если в результате будет получена «паутина взаимодействий» или архитектура будет асимметричной, вряд ли у вас выйдет что-то приличное. Скорее всего, вы почувствуете, правильна архитектура или нет.

Иногда область нестабильности может лежать за пределами системы. Например, хотя выплаты могут быть областью нестабильности из-за различных способов перечисления средств, система TradeMe как программный проект не направлена на реализацию платежной системы. Платежи играют вторичную роль по отношению к базовой ценности системы. Вероятно, система будет использовать ряд внешних платежных систем как *Ресурсы*. Каждый *Ресурс* может быть целой системой, обладающей собственными нестабильностями, но они выходят за рамки системы.

Команда проектирования составила следующий список областей, достаточно нестабильных для того, чтобы оказать влияние на архитектуру. В списке также представлены соответствующие компоненты архитектуры, инкапсулирующие области нестабильности:

- *Клиентские приложения*. Система должна предоставить возможность нескольким клиентским средам развиваться по отдельности и в разном темпе. Клиенты ориентированы на разные категории пользователей (мастера, подрядчики, операторы или учебные центры) или фоновые процессы (например, таймер, который периодически взаимодействует с системой). Клиентские приложения могут использовать разные технологии пользовательский интерфейса, устройства или API (например, учебный портал мог бы представлять собой простой API); они могут быть доступны локально или по интернету (мастера и операторы); пользователи могут подключаться или отключаться от них и т. д. Как и ожидалось, клиенты ассоциируются со значительной долей нестабильности. Каждая из нестабильных клиентских сред инкапсулируется в своем приложении *Клиент*.
- *Управление принадлежностью к системе*. Нестабильность присутствует в активностях добавления и удаления мастеров и подрядчиков и даже в по-

лучаемых ими премиях или скидках. Процедуры управления принадлежностью зависят от локального контекста и времени. Эти нестабильности инкапсулируются в *Менеджере принадлежности*.

- *Платежи*. Все возможные способы, которыми TradeMe может зарабатывать деньги, инкапсулированы в *Менеджере рынка*.
- *Проекты*. Требования и размеры проектов не только могут изменяться — они нестабильны и оказывают влияние на требуемое поведение. В больших и малых проектах могут использоваться разные потоки операций. Система инкапсулирует проекты в *Менеджере рынка*.
- *Споры*. При работе с людьми как минимум возникнет недопонимание; в худшем случае возможны попытки мошенничества. Нестабильность в процедурах разрешения споров инкапсулируется в *Менеджере принадлежности*.
- *Подбор и утверждение*. В этой области вступают в игру две нестабильности. Нестабильность поиска мастера, соответствующего потребностям проекта, инкапсулируется в *Ядре поиска*. Нестабильность критериев поиска и их определения инкапсулируется в *Менеджере рынка*.
- *Обучение*. Существует нестабильность в установлении соответствия между учебными курсами для мастеров и поиском доступного (или необходимого) курса. Управление нестабильностью процесса обучения инкапсулируется в *Менеджере обучения*. Поиск курсов и сертификатов инкапсулируется в *Ядре поиска*. Соответствие нормативной сертификации инкапсулируется в *Ядре нормативов*.
- *Нормативы*. Нормативы в любой стране могут измениться с течением времени. Кроме того, некоторые нормативы могут быть внутренними для компаний. Эта нестабильность инкапсулируется в *Ядре нормативов*.
- *Отчеты*. Все требования по предоставлению отчетов и аудиту, которым должна соответствовать система, инкапсулируются в *Ядре нормативов*.
- *Локализация*. С локализацией связаны две разные нестабильности. Элементы пользовательского интерфейса *Клиентов* инкапсулируют нестабильность языка и культуры. Для TradeMe ключевые участники решили, что это будет достаточно хорошим решением. В других случаях локализация может оказаться настолько сильной нестабильностью, что она заслуживает создания собственной подсистемы (например, *Менеджер* или *Ресурсы*). Локализация даже может повлиять на проектирование *Ресурсов*. Нестабильность законодательных норм в разных странах отражена в *Ядре нормативов*.
- *Ресурсы*. Ресурсы могут быть порталами для внешних систем (например, платежных); также они могут использоваться для хранения различных элементов (например, списков мастеров и проектов). Конкретная природа хранения нестабильна, от облачной базы данных до локального хранилища или отдельной полноценной системы.

- *Доступ к ресурсам.* Компоненты *Доступ к ресурсу* инкапсулируют нестабильность обращения к *Ресурсам* (например, местонахождение хранилища, его тип и технология доступа). Компоненты *Доступ к ресурсу* преобразуют атомарные бизнес-команды (например, для оплаты услуг мастера) в обращения к соответствующим *Ресурсам* (таким, как хранилища или платежные системы).
- *Модель развертывания.* Модель развертывания нестабильна. Иногда данные не могут выйти за границы географической области или же компания может захотеть развернуть систему в облаке (полностью или частично). Такие нестабильности инкапсулируются в композиции подсистем и вспомогательном компоненте *Шина сообщений*. Преимущества такого модульного паттерна взаимодействий в отношении операционных концепций системы описаны ниже.
- *Аутентификация и авторизация.* Система может выполнять аутентификацию *Клиентов* разными способами в зависимости от того, являются ли они пользователями и даже другими системами; также существуют разные варианты представления регистрационных и идентификационных данных. Авторизация практически не имеет ограничений, в ней множество способов хранения ролей или представления заявок. Эти нестабильности инкапсулируются во вспомогательном компоненте *Безопасность*.

Учтите, что между областями нестабильности и компонентами архитектуры нет однозначного соответствия. Например, в приведенном списке три области нестабильности соответствуют *Менеджеру рынка*. Вспомните, о чем говорилось в главе 3: *Менеджер* инкапсулирует нестабильность целого семейства логически связанных сценариев использования, а не отдельного сценария. В случае *Менеджера рынка* это управление проектом, установление соответствия между мастерами и проектами и взимание платы за подбор.

Слабые нестабильности

В архитектуре не отражены две дополнительные, более слабые области нестабильности:

- *Уведомление.* Подробности взаимодействия *Клиентов* с системой и взаимодействие системы с внешним миром могут быть нестабильными. Использование вспомогательного компонента *Шина сообщений* инкапсулирует эту нестабильность. Если в компании существует сильная потребность в неограниченных механизмах передачи данных (таких, как электронная почта или факс), возможно, *Менеджер уведомлений* был бы необходим.
- *Анализ.* Система TradeMe может проанализировать требования проектов, проверить запрашиваемых мастеров и даже предложить возможные кан-

дидатуры. Это позволяет TradeMe оптимизировать распределение мастеров между проектами. Система может анализировать проекты разными способами, причем такой анализ явно является областью нестабильности. При этом команда проектирования отвергла включение анализа как области нестабильности в архитектуру, потому что, как упоминалось ранее, оптимизация проектов не является основным профилем деятельности компании. Таким образом, предоставление оптимизаций относится к области спекулятивного проектирования. Весь необходимый анализ упаковывается в *Менеджер рынка*.

Статическая архитектура

На рис. 5.14 показано статическое представление архитектуры.

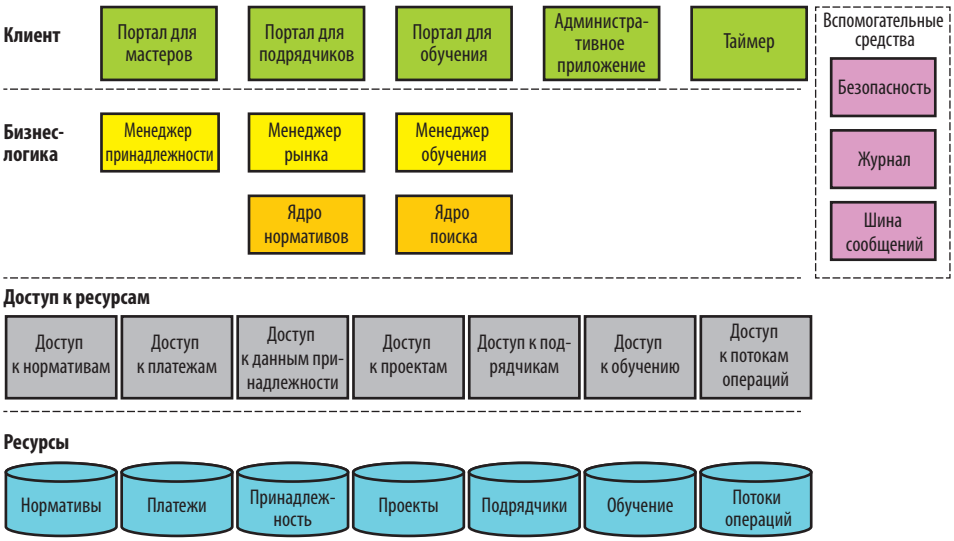


Рис. 5.14. Статическое представление архитектуры TradeMe

Клиенты

Клиентский уровень содержит портал для каждого типа участников, мастеров и подрядчиков. Также имеется портал для центра обучения, обеспечивающий выдачу и проверку регистрационных данных мастеров, и административное приложение, при помощи которого внутренние пользователи выполняют административные операции. Кроме того, клиентский уровень содержит внешние потоки операций, такие как планировщик или таймер, который периодически инициирует некоторое поведение в системе. Они включены в архитектуру для справки, но не являются частью системы.

ШИНА СООБЩЕНИЙ

Шина сообщений представляет систему «публикация/подписка» с очередью (рис. 5.15). Любое сообщение, публикуемое по шине, рассылается произвольному количеству подписчиков. Таким образом, шина сообщений предоставляет систему коммуникаций общего назначения с очередью *N:M*, где *N* и *M* могут быть любыми неотрицательными целыми числами. Если шина сообщений перестает работать или публикующая сторона отключается, то сообщения ставятся в очередь шины, а затем обрабатываются при восстановлении связи. Этот механизм обеспечивает доступность и ошибкозащищенность системы. Если подписчик (например, мобильное устройство) перестает работать или отключается, то сообщения публикуются в приватной очереди уровня подписчика, а затем обрабатываются, когда подписчик снова становится доступным. Если и публикатор, и подписчик подключены и доступны, то сообщения обрабатываются асинхронно.

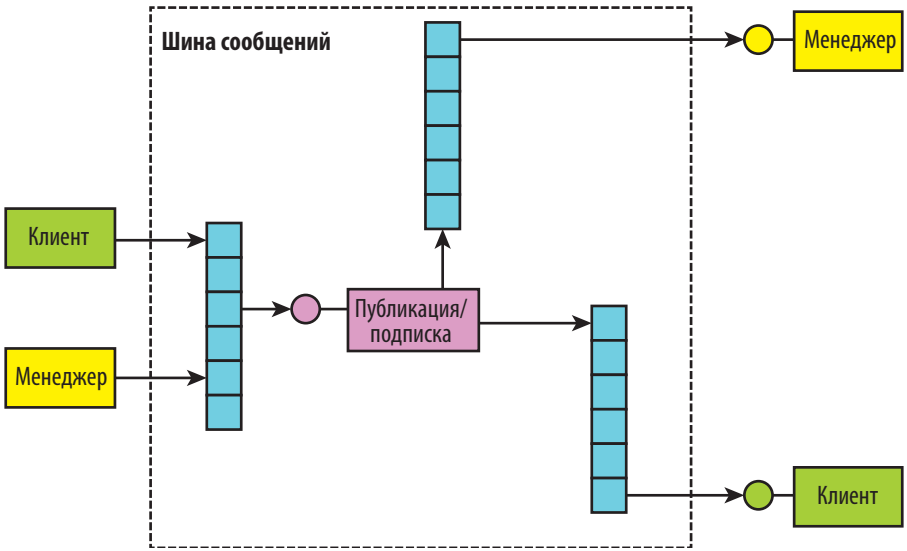


Рис. 5.15. Шина сообщений

Выбор технологии для шины сообщений не имеет отношения к архитектуре, а следовательно, выходит за рамки темы книги. Тем не менее конкретные функции, предоставляемые конкретной шиной сообщений, могут сильно повлиять на простоту реализации, поэтому правильный выбор потребует тщательных размышлений. Не все шины сообщений созданы равными, включая предложения от известных фирм-разработчиков. Как минимум шина сообщений должна поддерживать обработку отказов, обработку подозрительных сообщений, обработку транзакций, высокую пропускную способность, API уровня сервисов, поддержку

множественных протоколов (особенно не на базе HTTP) и надежную доставку сообщений. Также могут понадобиться и такие дополнительные функции, как фильтрация сообщений, анализ сообщений, расширенные средства перехвата, диагностика, автоматизация развертывания, упрощенная интеграция с хранилищем регистрационных данных и удаленная настройка конфигурации. Ни один из существующих продуктов не предоставит все эти функции. Чтобы снизить риск ошибочного выбора, следует начать с понятной, простой в использовании и бесплатной шины сообщений и реализовать архитектуру с этой шиной. Такая тактика позволит вам лучше понять нужные качества и атрибуты шины и определиться с приоритетами. Только тогда вы сможете выбрать лучший вариант, который действительно соответствует вашим потребностям.

Добавление шины сообщений в вашу архитектуру не отменяет необходимости установления архитектурных ограничений для паттернов коммуникаций. Например, взаимодействия «*Клиент-Клиент*» по шине должны быть запрещены.

Сервисы бизнес-логики

На уровне бизнес-логики находятся *Менеджер принадлежности* и *Менеджер рынка*, инкапсулирующие нестабильности, упоминавшиеся ранее. В двух словах, *Менеджер принадлежности* управляет нестабильностью при выполнении сценариев использования, связанных с принадлежностью к системе, а *Менеджер рынка* отвечает за сценарии использования, относящиеся к рынку. Учтите, что сценарии использования, относящиеся к принадлежности (например, добавление или удаление мастера), логически связаны друг с другом и при этом отличны от сценариев, относящихся к рынку (например, подбором мастера для проекта). *Менеджер обучения* инкапсулирует нестабильность выполнения сценариев использования, относящихся к повышению квалификации, например координации обучения и проверке сертификатов.

В системе существуют только два *Ядра*, инкапсулирующих некоторые нестабильности, перечисленные выше. *Ядро нормативов* инкапсулирует нестабильность нормативов и соответствия законодательству в разных странах и даже в одной стране в разное время. *Ядро поиска* инкапсулирует нестабильность при подборе кандидатур, который может осуществляться неограниченным множеством способов, от простого поиска по размеру оплаты до подбора с учетом факторов безопасности и учета показателей качества, применения искусственного интеллекта и методов машинного обучения.

Ресурсы и компоненты Доступ к ресурсу

Все данные, необходимые для управления рынком, — платежи, участники, проекты и т. д. — должны где-то храниться и иметь соответствующие компоненты

Доступ к ресурсу. Также предусмотрено хранилище данных потоков операций, о котором будет рассказано позднее.

Вспомогательные средства

Системе необходимы три вспомогательных компонента: *Безопасность*, *Шина сообщений* и *Журнал*. Все будущие вспомогательные средства (например, инструментальные средства управления) также будут размещаться на панели *Вспомогательные средства*.

Оперативные концепции

В системе TradeMe все коммуникации между всеми *Клиентами* и всеми *Менеджерами* осуществляются по *Шине сообщений* из категории *Вспомогательных средств*. Эта оперативная концепция изображена на рис. 5.16.

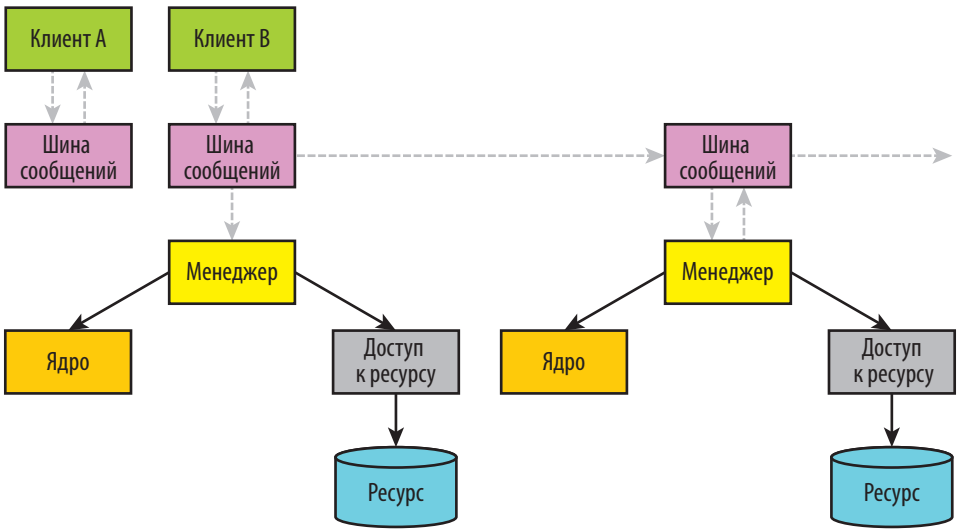


Рис. 5.16. Паттерн абстрактного взаимодействия в системе

В этом паттерне взаимодействий *Клиенты* и бизнес-логика в подсистемах отделяются друг от друга *Шиной сообщений*. Использование *Шины сообщений* обычно обеспечивает следующие оперативные концепции:

- Все коммуникации используют общий носитель (*Шина сообщений*). Этот способ передачи инкапсулирует природу сообщений, местонахождение сторон и коммуникационный протокол.

- Инициатор сценария использования (например, *Клиент*) и исполнитель сценария (например, *Менеджер*) никогда не взаимодействуют напрямую. Если они не знают о существовании друг друга, это позволяет им эволюционировать по отдельности, что способствует расширяемости.
- В одном сценарии использования могут быть задействованы разные параллельные *Клиенты*, каждый из которых выполняет свою часть сценария. Между Клиентами и системами не поддерживается выполнение с жестким параллелизмом. В свою очередь, это ведет к разделению временных шкал и ослаблению связей компонентов на временной шкале.
- Высокая пропускная способность обеспечивается благодаря тому, что очереди, лежащие в основе *Шины сообщений*, могут получать очень большое количество сообщений в секунду.

Сообщения и приложения

Оперативные концепции, поддерживаемые шиной сообщений, безусловно, полезны, но сами по себе они не оправдывают возрастающей сложности. Главная причина для выбора шины сообщений — поддержка самой важной оперативной концепции TradeMe: паттерна проектирования «Сообщение и есть Приложение».

При использовании этого паттерна проектирования никакого «приложения» нет. Нет набора компонентов или сервисов, на которые можно указать и определить как приложение. Вместо этого система включает нежесткий набор сервисов, которые публикуют и получают сообщения друг от друга (по шине сообщений, хотя это вторичный фактор). Эти сообщения связаны друг с другом. Каждый сервис, обрабатывающий сообщение, выполняет некоторую единицу работы, а затем отправляет сообщение обратно по шине. Другие сервисы проверяют сообщения, и некоторые из них (а может, один из них или ни одного) решают что-то сделать. Фактически сообщение, отправленное одним сервисом, заставляет другой сервис сделать нечто неизвестное для сервиса-отправителя. Таким образом ослабление связей доводится почти до предела.

Часто одно логическое сообщение может проходить через все сервисы. Следует учитывать, что сервисы могут добавить дополнительную контекстную информацию в сообщение (например, в заголовки), изменить предыдущий контекст, передать контекст из старого сообщения в новый и т. д. При таком подходе сервисы работают как функции преобразования сообщений. Основной аспект паттерна «Сообщение и есть Приложение» заключается в том, что требуемое поведение приложения является совокупностью этих преобразований и локальной работы, выполняемой отдельными сервисами. Любые требуемые изменения поведения порождают изменения в реакции ваших сервисов на сообщения, а не в архитектуре или в сервисах.

ПЕРСПЕКТИВНОЕ ПРОЕКТИРОВАНИЕ

Применение детализированных сервисов, интегрированных по шине сообщений на базе паттерна «Сообщение и есть Приложение», — один из лучших способов подготовки системы к будущему. Под «подготовкой системы к будущему» я имею в виду следующую эпоху программирования, применение модели акторов. Скорее всего, за следующее десятилетие отрасль разработки примет концепцию чрезвычайно детализированного использования сервисов, которые называются акторами. Хотя акторы являются сервисами, это очень простые сервисы. Акторы входят в граф или сеть акторов, и взаимодействуют они друг с другом только с использованием сообщений. Полученная сеть акторов может выполнять вычисления или хранить данные. Программа не является совокупностью кода акторов; вместо этого программа или требуемое поведение образуется из перемещения сообщений по сети. Чтобы изменить программу, вы изменяете сеть акторов, а не сами акторы.

Построение системы таким способом предоставляет такие фундаментальные преимущества, как улучшенное соответствие реальным бизнес-моделям, высокая степень параллелизма без блокировок и также возможность построения систем, выходящих за рамки нынешних возможностей, — интеллектуальных сетей электропередачи, систем командования и контроля, а также обобщенного искусственного интеллекта. Использование современных технологий и платформ с принципом «Сообщение и есть Приложение» очень сильно ориентировано на модель акторов. Например, в TradeMe мастера и подрядчики являются акторами. Проекты являются сетями акторов, а другие акторы (такие, как Менеджер рынка) образуют сеть. Принятие архитектуры TradeMe сегодня подготовило компанию к будущему без ущерба для текущего состояния дел[1].

Бизнес-цели TradeMe оправдывают применение этого паттерна из-за требуемой расширяемости. Компания может расширить систему, добавляя сервисы обработки сообщений, тем самым избегая модификации существующих сервисов и риска для работающей реализации. Тем самым обеспечивается правильная поддержка директивы из главы 3, которая гласит, что системы всегда должны строиться инкрементно, а не итеративно. Также здесь хорошо выполняется цель перспективного проектирования, потому что ничто в этом паттерне не привязывает систему к текущим требованиям. Паттерн также открывает возможность элегантной интеграции с внешними системами — еще одна бизнес-цель.

Как это всегда бывает, реализация этого паттерна не дается бесплатно. Не каждая организация может обосновать применение этого паттерна или даже наличие шины сообщений. Плата почти всегда воплощается в форме допол-

[1] Дополнительную информацию о модели акторов см. Juval Lowy, Actors: The Past and Future of Software Engineering (YouTube/IDesignIncTV, 2017).

нительной сложности системы, новых подвижных частей, новых API, которые приходится изучать, проблем развертывания и безопасности, нетривиальных сценариев отказов и т. д. К положительным сторонам следует отнести то, что вы получаете систему с изначальной слабой связанностью, ориентированную на преодоление пересмотра требований, расширяемость и повторное использование. В общем случае этот паттерн следует применять тогда, когда вы можете вложить средства в платформу и располагаете поддержкой вашей организации. Во многих случаях упрощенная архитектура, в которой *Клиенты* только ставят в очередь вызовы к *Менеджерам*, могла бы лучше подойти для команды разработки. Всегда выверяйте свою архитектуру по способностям и зрелости разработчиков и руководства. В конце концов, трансформировать архитектуру намного проще, чем преобразовать организацию. А после того, как организационные возможности сформируются, вы можете реализовать полноценный паттерн «Сообщение и есть Приложение».

Менеджеры потоков операций

В Методе нестабильность бизнес-потоков операций инкапсулируется в *Менеджерах*. Ничто не мешает вам просто программировать потоки операций в *Менеджерах*, а затем, когда потоки операций изменятся, изменять код *Менеджеров*. Проблема такого подхода заключается в том, что нестабильность в потоках операций может превысить возможности разработчиков в отношении затрат времени и усилий для своевременного обновления кода.

Следующая оперативная концепция TradeMe — использование *Менеджеров* потоков операций. Я вскользь упоминал эту концепцию в главе 2 при обсуждении системы торговли акциями, но в этой главе она будет систематизирована в оперативный паттерн. Все *Менеджеры* в TradeMe являются *Менеджерами потока операций* — сервисами, которые позволяют создавать, хранить, читать и выполнять потоки операций. Теоретически это всего лишь очередной *Менеджер*. Однако на практике такие *Менеджеры* почти всегда используют те или иные сторонние инструменты для выполнения и хранения потоков операций. Для каждого вызова *Клиента Менеджер* потока операций загружает не только поток операций правильного типа, но и его конкретный экземпляр с конкретным состоянием и контекстом, выполняет поток операций и снова сохраняет его в хранилище потоков. Загрузка и состояние экземпляра потока операций поддерживает возможность создания продолжительных потоков. *Менеджеру* также не нужно поддерживать какую-либо разновидность сеансов с *Клиентом* для поддержания состояния. Разные вызовы от одного пользователя в одном выполнении потока операций могут поступать с разных устройств по разным подключениям, но при этом они содержат уникальный идентификатор экземпляра потока операций, который *Менеджер* должен загрузить и выполнить, а также информацию о клиенте (например, его адрес, то есть URI).

ВЫБОР СРЕДСТВ УПРАВЛЕНИЯ ПОТОКАМИ ОПЕРАЦИЙ

Выбор технологии для управления потоками операций не имеет прямого отношения к архитектуре, поэтому он в книге не рассматривается. Тем не менее если архитектура этого требует, вам следует выбрать правильный инструмент управления потоком. Существуют буквально десятки решений, причем разные средства обладают сильно различающейся функциональностью. Как минимум инструмент управления должен поддерживать визуальное редактирование потоков операций, сохранение и восстановление экземпляров потоков операций, вызов сервисов из потоков операций по разным протоколам, отправку сообщений по шине сообщений, предоставление потоков операций как сервисов по разным протоколам, вложение потоков операций, создание библиотек потоков операций, определение общих шаблонов повторяющихся потоков операций, которые могут настраиваться позднее, и отладку потоков операций. Также желательно иметь возможность воспроизведения, оркестровки и профилирования потоков операций, а также интеграции их с системой диагностики.

Чтобы добавить или изменить функцию, вы просто добавляете или изменяете потоки операций задействованных *Менеджеров*, при этом можно обойтись без изменения реализации отдельных задействованных сервисов. Это способ предоставления функций как аспектов интеграции (см. главу 4) и материальный аспект формулировки миссии системы, позволяющий продемонстрировать поддержку архитектуры со стороны бизнеса.

Настоящая необходимость в использовании *Менеджеров* потоков операций возникает тогда, когда система должна справляться с высокой нестабильностью. С *Менеджером* потока операций вы просто вносите изменения в требуемое поведение и развертываете вновь сгенерированный код. Природа таких изменений тесно связана с инструментарием потоков операций, которым вы пользуетесь. Например, одни инструменты используют редакторы сценариев, тогда как другие используют визуальные потоки операций, которые выглядят как диаграммы активности и генерируют или даже развертывают код потока операций.

Вы даже можете предоставить возможность владельцам продукта или конечным пользователям редактировать требуемое поведение (конечно, с принятием необходимых защитных мер). Такой подход радикально сокращает время внедрения новых функций, а команда разработки продукта может сосредоточиться на базовых сервисах вместо того, чтобы гоняться за изменениями в требованиях.

Бизнес-потребности TradeMe оправдывают применение этого паттерна, потому что цель быстрого внедрения новых функций не может быть достигнута с ручным программированием в маленькой распределенной команде. Применение *Менеджера* потока операций открывает широкие возможности адаптации для разных рынков, удовлетворяя другую цель системы.

И снова следует тщательно оценить, насколько эта концепция применима к вашему конкретному случаю. Убедитесь в том, что уровень нестабильности потока операций оправдывает дополнительную сложность, время обучения и изменения в процессе разработки.

Проверка архитектуры

Прежде чем начинать работу, необходимо иметь твердую уверенность в том, что проектировочное решение сможет поддерживать необходимое поведение. Как объясняется в главе 4, для проверки решения необходимо показать, что оно может поддерживать базовые сценарии использования за счет интеграции разных областей нестабильности, инкапсулированных в ваших сервисах. Подтверждение осуществляется демонстрацией соответствующих цепочек вызовов или диаграммы последовательности для каждого сценария использования. Для завершения сценария использования может потребоваться более одной диаграммы.

Важно продемонстрировать действительность вашего проектировочного решения не только себе, но и другим. Если вы не можете подтвердить вашу архитектуру или результат проверки оказывается слишком неоднозначным, необходимо вернуться к проектированию.

Как упоминалось ранее, среди немногочисленных сценариев использования, предоставленных компанией для TradeMe, присутствовал всего один кандидат для базового сценария использования: подбор мастера. Архитектура TradeMe была модульной и изолированной от всех сценариев использования до такой степени, что команда проектирования могла продемонстрировать поддержку всех предоставленных сценариев использования, не только базового сценария подбора мастера. В следующем разделе продемонстрирована проверка сценариев использования TradeMe и оперативной концепции новой системы.

Добавление мастера/подрядчика

Сценарий использования «Добавление мастера/подрядчика» включает несколько областей нестабильности: клиентские приложения мастера (или подрядчика), поток операций для добавления нового участника, соответствие правовым нормам и используемая платежная система. Сценарий использования на рис. 5.1 можно упорядочить и упростить, добавив на диаграмму дорожки, как показано на рис. 5.17.

На рис. 5.17 показано, что выполнение сценария использования требует взаимодействия между приложением-*Клиентом* и подсистемой принадлежности. Это с очевидностью следует из цепочек вызовов на рис. 5.18 (сценарий использования для добавления подрядчика идентичен, но в нем используется приложение для подрядчика — *Портал для подрядчиков*). В соответствии с оперативными

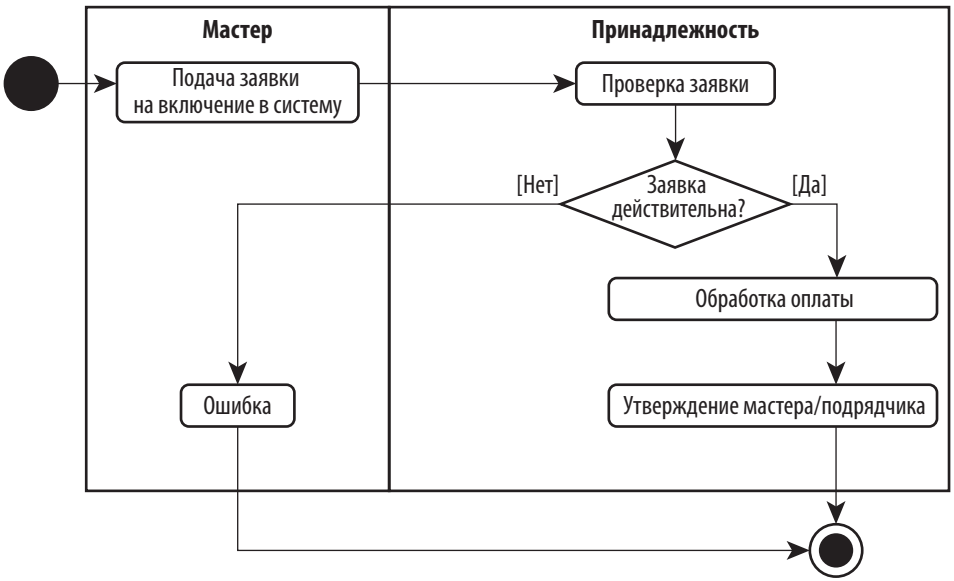


Рис. 5.17. Сценарий использования «Добавление мастера/подрядчика» с дорожками

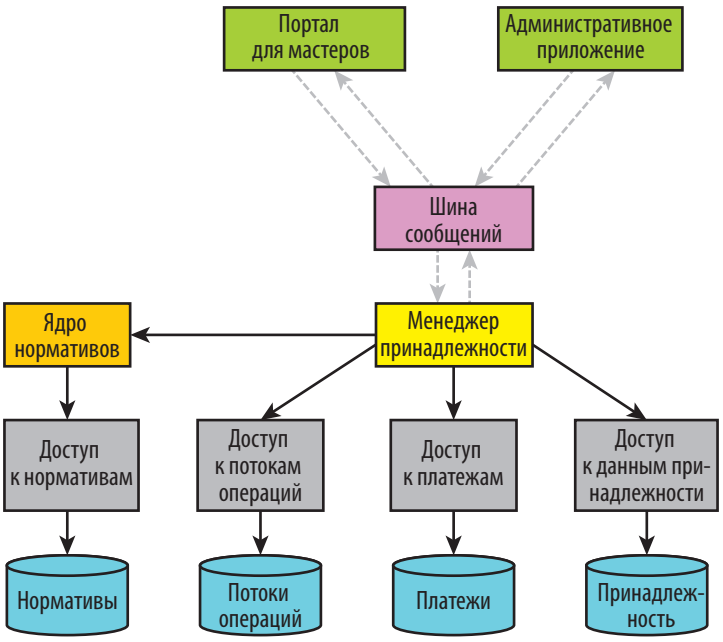


Рис. 5.18. Цепочка вызовов при добавлении мастера/подрядчика

концепциями TradeMe на рис. 5.18 приложение-*Клиент* (в данном случае либо *Портал для мастеров*, где непосредственно подается заявка на участие, либо *Административное приложение*, в котором оператор может добавить участника вручную) отправляет запрос *Шине сообщений*.

При получении сообщения *Менеджер принадлежности* (который является *Менеджером* потока операций) загружает соответствующий поток операций из хранилища. При этом он либо запускает новый поток операций, либо восстанавливает существующий для продолжения выполнения потока. После того как поток операций завершит выполнение запроса, *Менеджер принадлежности* отправляет обратно по *Шине сообщений* сообщение, обозначающее новое состояние потока операций — например, завершение или признак того, что другой *Менеджер* может теперь начать обработку потока операций, перешедшего в новое состояние. *Клиенты* также могут отслеживать *Шину сообщений* и оповещать пользователей о статусе их запросов. *Менеджер принадлежности* обращается к *Ядру нормативов*, которое проверяет мастера или подрядчика, добавляет мастера или подрядчика в хранилище *Участников* и уведомляет *Клиентов* по *Шине сообщений*.

Запрос мастера

В сценарии использования «Запрос Мастера» представляют интерес две области: подрядчик и рынок (рис. 5.19). После исходной проверки запроса этот сценарий использования инициирует выполнение другого сценария, «Подбор мастера».

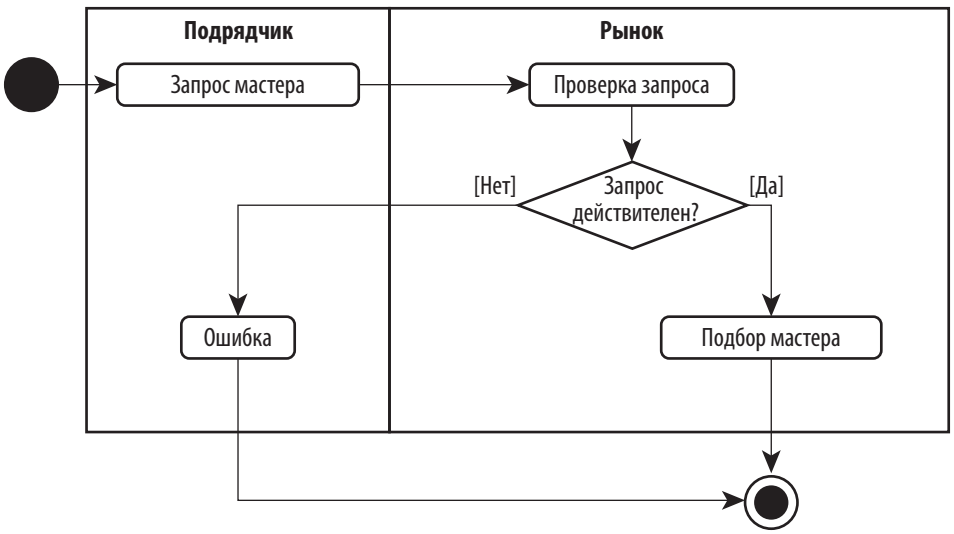


Рис. 5.19. Сценарий использования «Запрос мастера» с дорожками

Цепочки вызовов изображены на рис. 5.20. Клиенты, такие как *Портал для подрядчиков* или внутренний пользователь *Административного приложения*, отправляют по шине сообщение с запросом мастера. *Менеджер рынка* получает это сообщение, загружает поток операций, соответствующий запросу, и выполняет такие действия, как обращение к *Ядру нормативов* за информацией, которая может относиться к этому запросу, или обновление проекта запросом мастера. Затем *Менеджер рынка* может отправить по *Шине сообщений* обратное сообщение о том, что кто-то запрашивает мастера. Это приведет к запуску потоков операций подбора и назначения, разделенных на временной шкале.

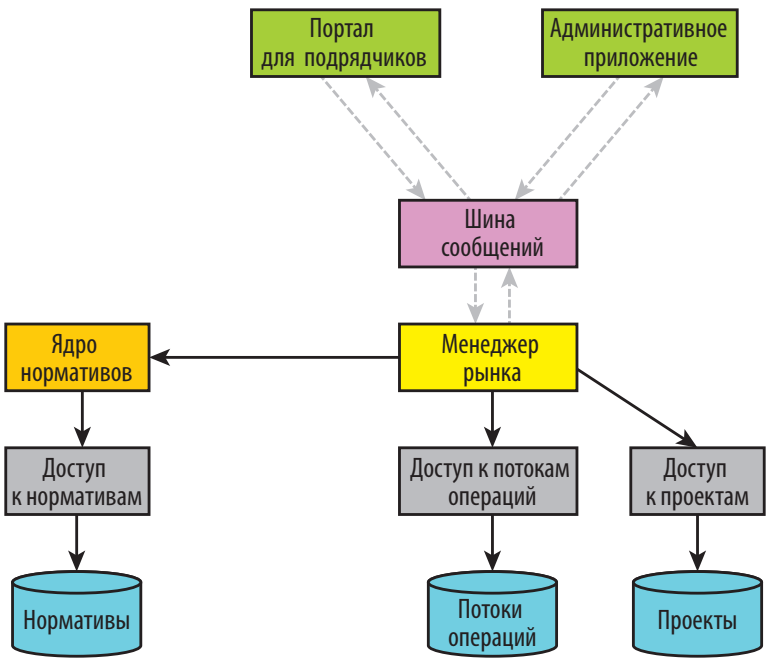


Рис. 5.20. Цепочки вызовов при запросе мастера (до подбора)

Подбор мастера

В базовом сценарии использования «Подбор мастера» представляют интерес несколько областей. Первая — инициатор запроса мастера, по которому был запущен сценарий подбора мастера. Инициатором может быть *Клиент* (подрядчик или операторы цифровой платформы), как на рис. 5.20, но это также может быть таймер или любая другая подсистема, которая запускает поток операций по подбору мастера. Другие области, представляющие интерес, — рынок, нормативы, поиск и в конечном итоге принадлежность (рис. 5.21).

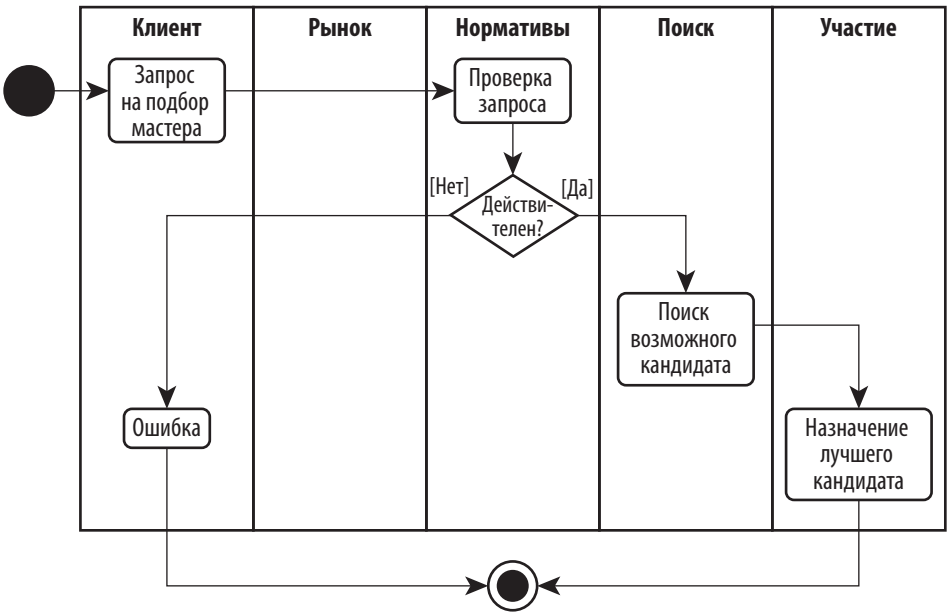


Рис. 5.21. Сценарий использования «Подбор мастера» с дорожками

После того как вы поймете, что нормативы и поиск являются элементами рынка, диаграмму активности можно переработать так, как показано на рис. 5.22. Таким образом обеспечивается простое установление соответствия со структурой подсистем.

На рис. 5.23 изображена соответствующая цепочка вызовов. И снова эта цепочка вызовов симметрична с другими цепочками вызовов в том смысле, что первым действием должна быть загрузка соответствующего потока операций и его выполнение. Последний вызов в цепочке вызовов к *Шине сообщений* и *Менеджеру принадлежности* запускает сценарий использования «Назначение мастера».

Обратите внимание на компонуемость этой архитектуры. Предположим, компании требуется справиться с серьезной нестабильностью при анализе потребностей проекта. Цепочка вызовов при поиске кандидата позволяет отделить поиск от анализа. Вы просто добавляете *Ядро анализа* для инкапсуляции отдельного набора алгоритмов анализа.

Система TradeMe даже может использоваться в целях бизнес-аналитики для получения ответов на вопросы типа: «Могли ли мы сделать что-то лучше?» Например, цепочка вызовов вроде изображенной на рис. 5.23 может использоваться для анализа намного более сложного сценария «Анализ всех

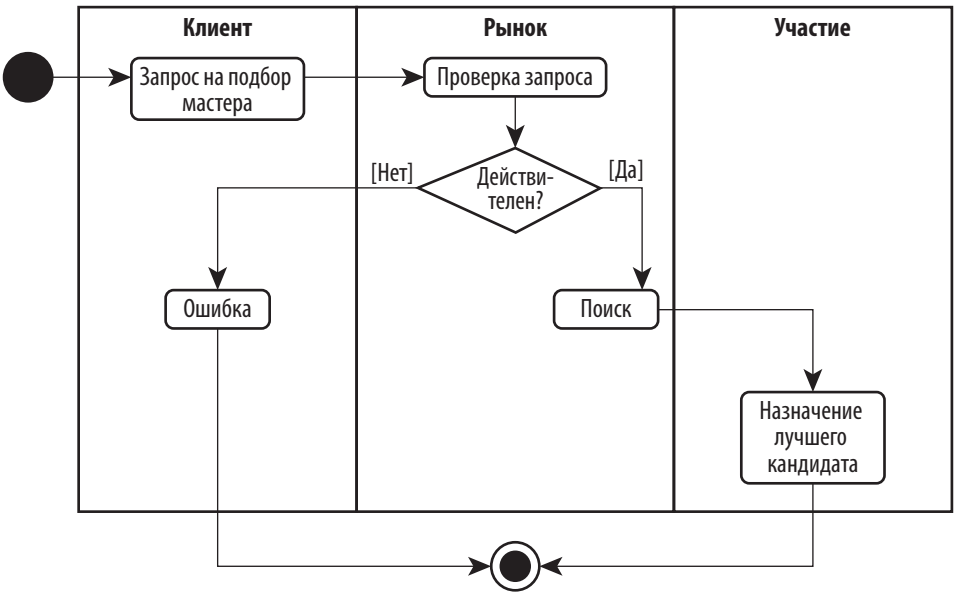


Рис. 5.22. Переработанные дорожки для сценария использования «Подбор мастера»

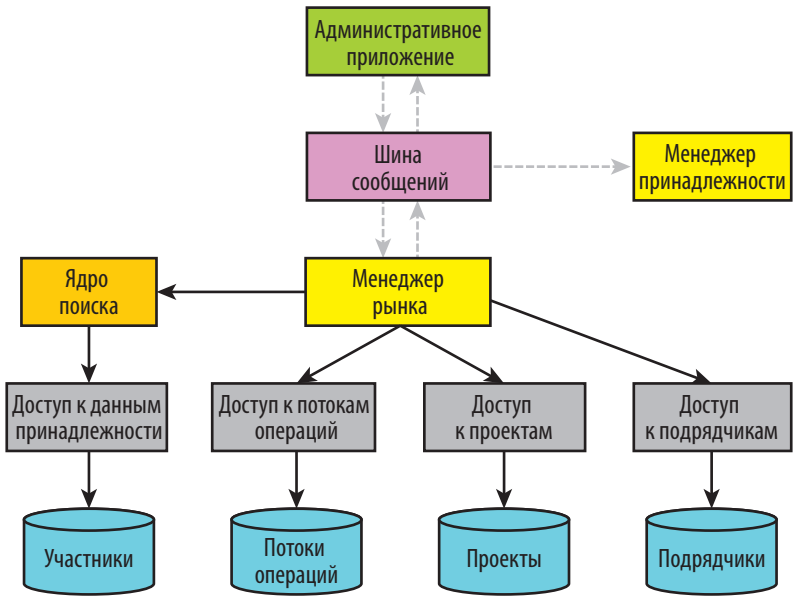


Рис. 5.23. Цепочки вызовов для сценария использования «Подбор мастера»

проектов с 2016 по 2019 г.», причем архитектура компонентов при этом бы нисколько не изменилась. Вероятно, таких сценариев использования можно придумать сколько угодно, и в этом заключается вся суть: вы создали незамкнутое проектировочное решение, которое может быть расширено для реализации любых из этих будущих сценариев, — по-настоящему компонуемая архитектура.

Назначение мастера

В сценарии использования «Назначение мастера» задействованы четыре важные области: клиент, принадлежность, нормативы и рынок. Учтите, что сценарий использования не зависит от того, кто инициировал его выполнение, будь то реальный внутренний пользователь или простое сообщение-запрос, полученное по шине сообщений от другой подсистемы. Например, сценарий использования «Подбор мастера» мог бы запустить сценарий назначения как непосредственное продолжение потока операций при автоматическом подборе и назначении.

После переработки диаграммы активности легко устанавливается соответствие с подсистемами (рис. 5.25).

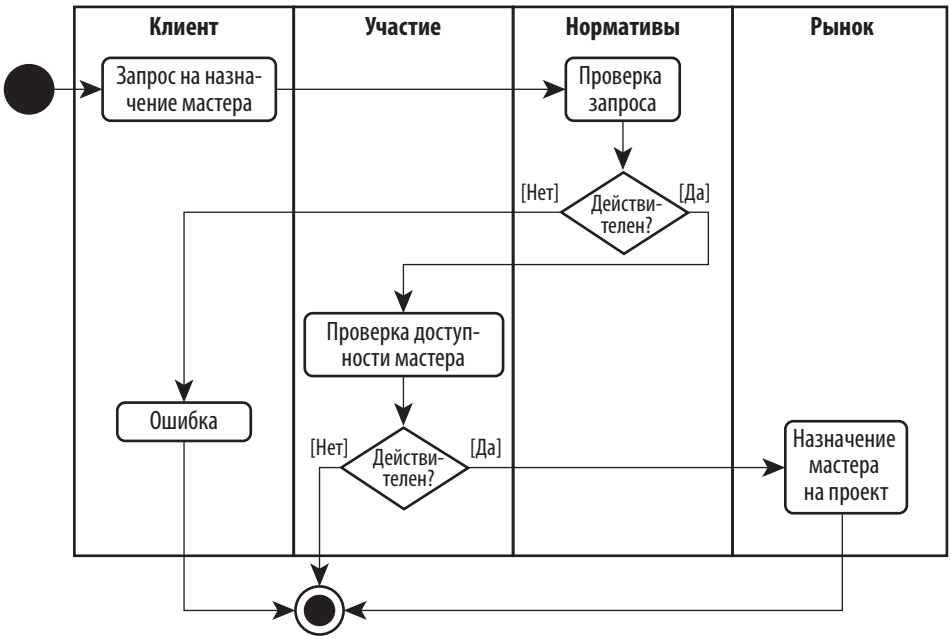


Рис. 5.24. Сценарий использования «Назначение мастера» с дорожками

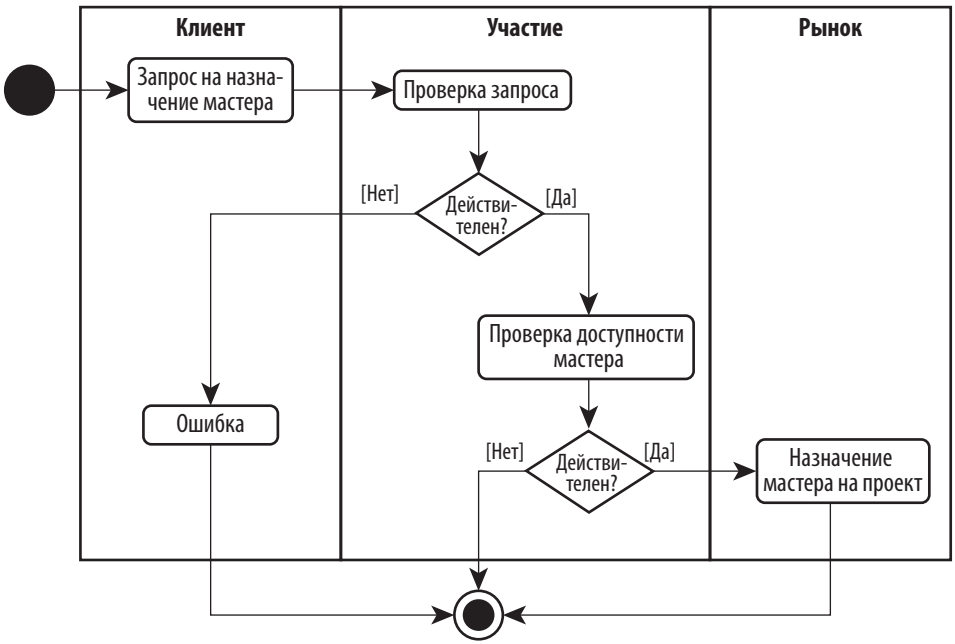


Рис. 5.25. Объединение дорожек сценария использования «Назначение мастера»

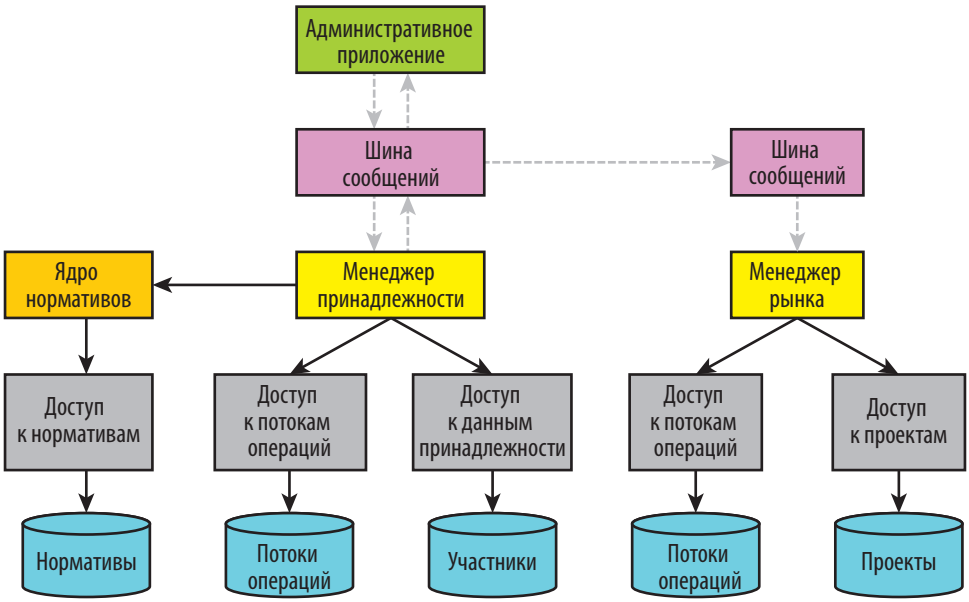


Рис. 5.26. Цепочки вызовов для сценария использования «Назначение мастера»

Как и во всех предыдущих цепочках вызовов, на рис. 5.26 показано, как *Менеджер принадлежности* выполняет поток операций, который в конечном итоге приводит к назначению мастера на проект. Эта работа выполняется совместно *Менеджером принадлежности* и *Менеджером рынка*, каждый из которых управляет соответствующей подсистемой. Учтите, что *Менеджер принадлежности* ничего не знает о *Менеджере рынка* — он всего лишь отправляет сообщение по шине. *Менеджер рынка* получает это сообщение и обновляет проект в соответствии со своим внутренним потоком операций. В свою очередь, *Менеджер рынка* может отправить другое сообщение по *Шине сообщений*, чтобы запустить другой сценарий использования — например, генерирование отчета по проекту, или запрос оплаты у подрядчика, или что-нибудь другое. В этом заключается вся суть паттерна проектирования «Сообщение и есть Приложение»: логическое сообщение «назначения» перемещается между сервисами, инициируя локальное поведение в процессе перемещения. *Клиент* также может отслеживать *Шину сообщений* и оповестить пользователя о том, что назначение находится в процессе выполнения.

Завершение найма

В предыдущих сценариях использования дорожки на исходной диаграмме включали область нормативов, которая в дальнейшем была консолидирована в подсистему принадлежности. Так как эта закономерность повторялась, на рис. 5.9 показана переработанная диаграмма для сценария использования «Завершение найма». На диаграмме различия обозначены достаточно четко, что позволяет установить явные соответствия в архитектуре.

На рис. 5.27 показана цепочка вызовов для завершения найма мастера. *Менеджер рынка* инициирует поток операций завершения и уведомляет *Менеджера принадлежности* о завершении.

Любое состояние ошибки или отклонение от «благополучного пути» добавляет пунктирную серую стрелку от *Менеджера принадлежности* к *Шине сообщений* и в конечном итоге обратно к *Клиенту*. На рис. 5.28 изображена диаграмма последовательности, демонстрирующая это взаимодействие, без вызовов между сервисами *Доступ к ресурсу* и *Ресурсы*.

Наконец, диаграмма цепочки вызовов на рис. 5.27 (или диаграмма последовательности на рис. 5.28) предполагает, что сценарий использования завершения будет инициирован при завершении проекта, а подрядчик уволит назначенного мастера. Но сценарий также может быть инициирован мастером, отправляющим сообщение от *Портала для мастеров* к *Менеджеру принадлежности*, что приведет к прохождению цепочки вызовов в обратном направлении (от *Менеджера принадлежности* к *Менеджеру рынка* и приложениям-*Клиентам*). Это в очередной раз демонстрирует гибкость архитектуры.

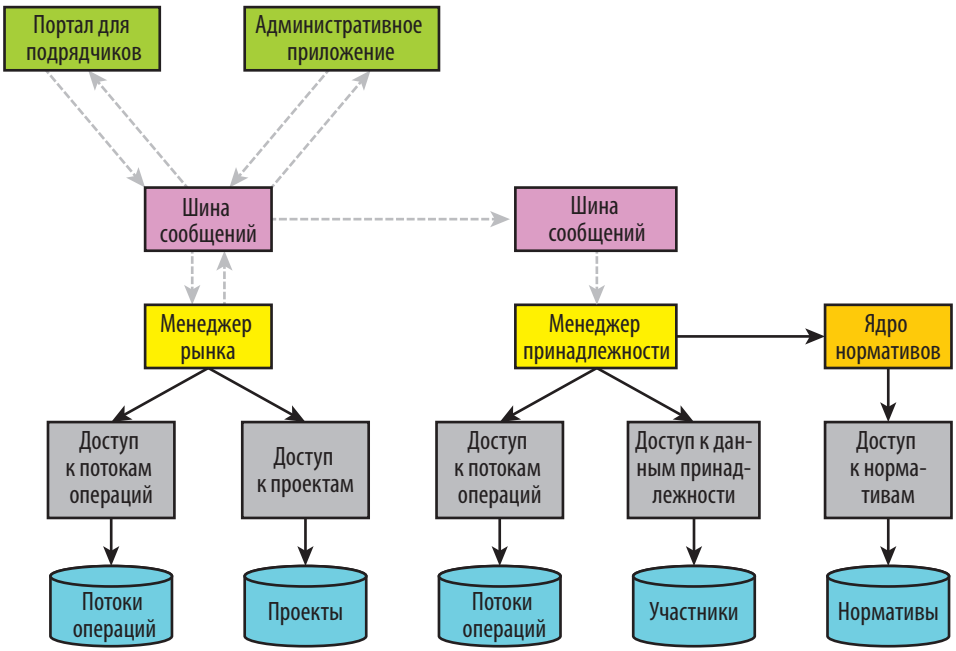


Рис. 5.27. Цепочки вызовов для сценария использования «Завершение найма»

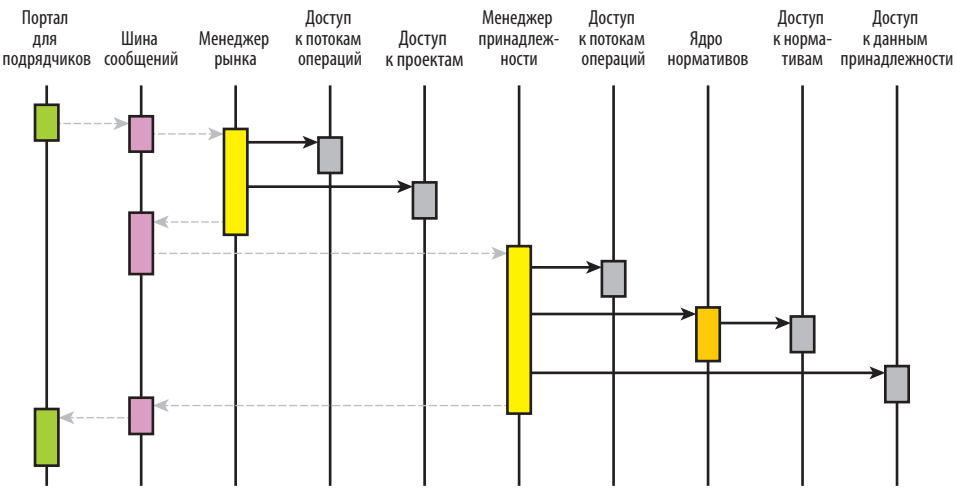


Рис. 5.28. Диаграмма последовательности для сценария использования «Завершение найма»

Оплата мастера

Остальные сценарии использования точно следуют схеме взаимодействий и паттерну проектирования сценариев, описанным выше, поэтому здесь будут приведены только краткие описания. Также обратите внимание на высокую степень самоподобия, или симметрии, в цепочках вызовов. На рис. 5.6 показан сценарий использования «Оплата мастера», а подтверждающая цепочка вызовов изображена на рис. 5.29.

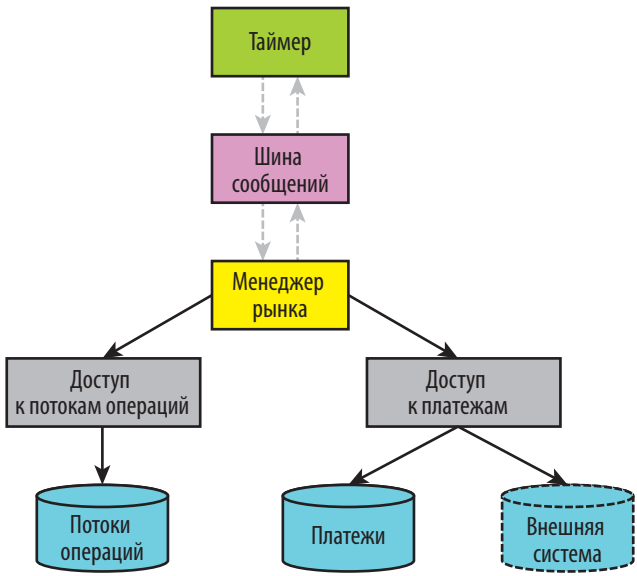


Рис. 5.29. Цепочки вызовов для сценария использования «Оплата мастера»

В отличие от предыдущих цепочек вызовов, оплата инициируется планировщиком или таймером. Планировщик отделен от непосредственных компонентов и ничего не знает о внутреннем устройстве системы: все, что он делает, — это отправка сообщения по шине. Непосредственная оплата осуществляется компонентом *Доступ к платежам* при обновлении хранилища *Платежей* и обращении к внешней платежной системе, которая является *Ресурсом* для TradeMe.

Создание проекта

В другом простом сценарии использования *Менеджер рынка* реагирует на запрос на создание проекта выполнением соответствующего потока операций (рис. 5.30 и диаграмма сценария использования на рис. 5.7). Независимо от

того, сколько шагов для этого потребуется или какое количество ошибок может быть задействовано в создании, сама природа паттерна «*Менеджер* потока операций» позволяет использовать столько перестановок, сколько потребуется.

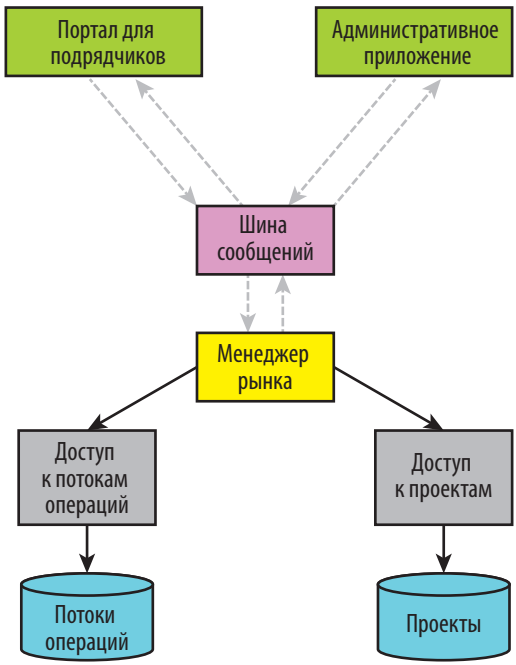


Рис. 5.30. Цепочки вызовов для сценария использования «Создание проекта»

Закрытие проекта

В сценарии использования «Закрытие проекта» задействован как *Менеджер рынка*, так и *Менеджер принадлежности* (рис. 5.31 и сценарий использования на рис. 5.8). И снова TradeMe решает эту задачу взаимодействием между этими двумя основными абстракциями; взаимодействие не отличается от изображенного на рис. 5.27.

> **ПРИМЕЧАНИЕ** В главе 13 продемонстрировано проектирование проекта для TradeMe, основанное на описании архитектуры из этой главы. Как и при описании проектирования системы, в этом подробном практическом примере перебираются различные варианты для определения плана проекта, обеспечивающего оптимальную комбинацию времени, затрат и рисков для проекта.

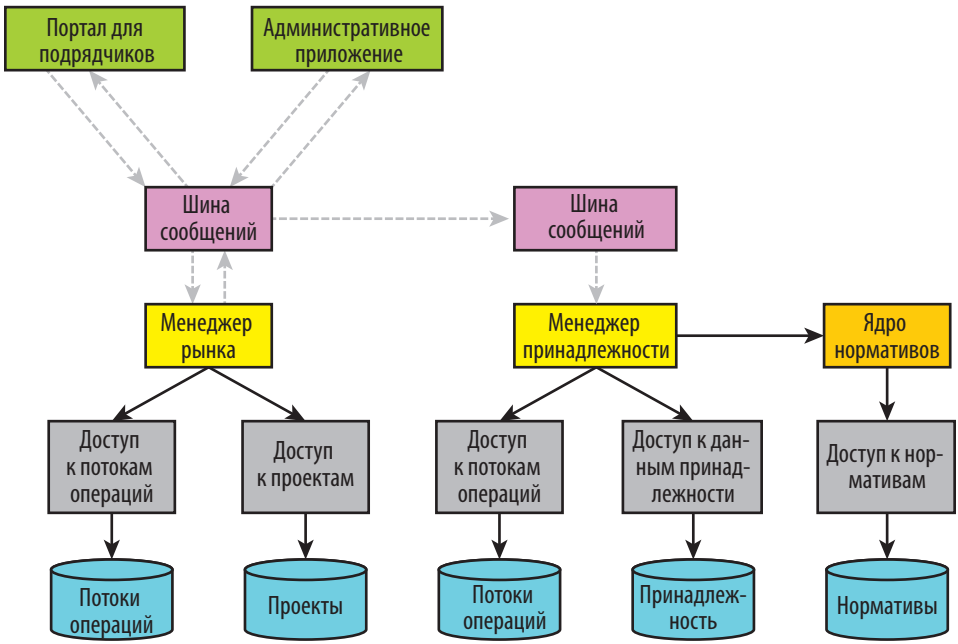


Рис. 5.31. Цепочки вызовов для сценария использования «Закрытие проекта»

Что дальше?

Этот длинный пример проектирования системы завершает часть I книги. Наличие готового проектировочного решения — всего лишь первый ингредиент успеха. Затем наступает очередь планирования проекта. Куйте железо, пока горячо: проектирование системы всегда должно сопровождаться планированием проекта. В идеале это должно происходить без перерыва, как единая работа по проектированию.

ЧАСТЬ II
Планирование проекта

6 Мотивация

Проектированием программной системы дело не ограничивается — вы также должны создать проект для построения системы. В процессе планирования необходимо точно вычислить оценку продолжительности и стоимости, предусмотреть несколько хороших вариантов исполнения, спланировать ресурсы и даже проверить ваш план, чтобы убедиться в его разумности и жизнеспособности. Планирование проекта требует понимания зависимостей между сервисами и активностями, критического пути интеграции, правильного распределения персонала и заложенных рисков. Все эти проблемы исходят от проектирования системы, и правильное их решение является технической задачей. А значит, планированием проекта должны заниматься вы, архитектор программных систем, как ответственный технический специалист.

Планирование проекта должно рассматриваться как продолжение работы по проектированию системы. Объединение проектирования системы и планирования проекта создает нелинейный эффект, который кардинально повышает вероятность успеха проекта. Также важно отметить, что планирование проекта не является составной частью управления проектом. Скорее оно играет для управления проектом ту же роль, которую архитектура играет для программирования.

Вторая часть книги посвящена планированию проектов. В следующих главах представлены традиционные идеи, а также мои собственные, проверенные на практике методы и приемы, относящиеся ко всему накопленному опыту проектирования современных программных проектов. В этой главе представлена история вопроса и основная мотивация для планирования проекта.

Зачем нужно планирование проекта?

Ни один проект не располагает бесконечным временем, средствами или ресурсами. Все серьезные планы проектов всегда жертвуют временем ради экономии средств, или наоборот. Более того, для каждого конкретного проекта существу-

ют многочисленные возможные комбинации сроков и затрат. Если в вашем распоряжении один разработчик или четыре разработчика, если вам выделены два года или полгода, если вы пытаетесь свести к минимуму риски и довести до максимума вероятность успеха — у вас будут получаться разные проекты.

В ходе планирования проекта вы должны предоставить схему управления с несколькими приемлемыми вариантами компромиссов между сроками, затратами и рисками, которые бы позволяли руководству и другим лицам, принимающим решения, заранее выбрать вариант, лучше всего соответствующий их потребностям и ожиданиям. Предоставление вариантов при планировании проекта — ключ к успеху. Нахождение сбалансированного (и даже оптимального) решения является задачей, разрабатываемой на высоком техническом уровне. Я говорю «разрабатываемой» не только из-за необходимых вычислений и проектирования, но и потому, что суть инженерной разработки заключается в нахождении компромиссов и учете реальной ситуации.

К трудностям планирования проекта добавляется тот факт, что единственно правильного решения не существует даже для одного набора ограничений, поскольку для любой системы всегда существует много разных подходов к проектированию. Проекты, спланированные с расчетом на выполнение сжатого графика, будут сопряжены с гораздо большим риском и сложностью, чем проекты, спланированные для снижения затрат и минимизации риска. Не существует «Истинного Проекта»; есть только бесчисленные варианты. Ваша задача — сузить этот спектр почти бесчисленных возможностей до нескольких хороших критериев проектирования проекта вроде следующих:

- Самый экономичный вариант построения системы.
- Самый быстрый вариант выпуска системы.
- Самый безопасный вариант выполнения обязательств.
- Лучшая комбинация сроков, затрат и риска.

В следующих главах мы покажем, как узнать хорошие варианты планирования проекта. Если вы не предоставите эти варианты, то за конфликты с руководством вам не стоит винить никого, кроме себя. Сколько раз вы трудились над проектированием системы и выдавали его руководству только для того, чтобы услышать в ответ: «В вашем распоряжении год и четыре разработчика»? Любые корреляции между годом, четырьмя разработчиками и тем, что в действительности потребуется для построения системы, непредсказуемы — как и ваши шансы на успех. Если вы представите ту же архитектуру с тремя-четырьмя вариантами планирования проекта — жизнеспособными, но отражающими разные соотношения сроков, затрат и риска — это в корне изменит всю динамику встречи. Теперь разговор будет вестись вокруг того, какой из этих вариантов стоит выбрать.

Вы должны сформировать среду, в которой руководители могут принимать хорошие решения. Для этого вы должны предлагать им только хорошие варианты. Тогда тот вариант, который они выберут, будет хорошим решением.

Планирование и адекватность проекта

Планирование проекта позволяет «осветить темные углы», то есть заранее получить представление об истинном масштабе проекта. Планирование проекта заставляет руководителей тщательно продумывать работу до ее начала, выявлять неожиданные отношения и ограничения, представлять все активности и распознавать разные варианты построения системы. Оно позволяет организации определить, хочет ли она браться за этот проект вообще. В конце концов, если истинные затраты и продолжительность превышают допустимые ограничения, зачем вообще начинать работу, раз проект все равно будет отменен после превышения сроков или затрат?

Когда план проекта будет завершен, вы исключите столь типичные оценки затрат, берущиеся с потолка, «смертельные марафоны» при разработке, самообман относительно успеха проекта и ужасающе дорогие серии проб и ошибок. После того как работа будет завершена, хорошо спланированный проект также закладывает основу, на базе которой ответственные руководители будут оценивать и продумывать эффект предлагаемых изменений для графика и бюджета.

Инструкции по сборке

Планирование проекта далеко не сводится к принятию правильных решений. Планирование проекта также служит своего рода инструкцией по сборке системы. Подумайте, стали бы вы покупать отлично спроектированную мебель из IKEA без инструкции по сборке? Какой бы удобной ни была эта мебель, никому не захочется самостоятельно разбираться в том, где и в каком порядке нужно устанавливать десятки винтов, гаек и пластин.

Ваша программная система существенно сложнее мебели, однако архитекторы нередко полагают, что разработчики и менеджеры проекта просто возьмутся за построение проекта и как-нибудь разберутся по ходу дела. Понятно, что этот ситуативный подход не является самым эффективным вариантом сборки системы. Как будет показано в следующих главах, планирование проекта изменяет такую ситуацию, потому что чтобы определить, сколько времени и средств потребуется на работу над проектом, необходимо сначала разобраться в том, как именно вы собираетесь ее строить. А следовательно, к каждому варианту планирования проекта должен прилагаться собственный набор инструкций по сборке.

Иерархия потребностей

В 1943 году Абрахам Маслоу опубликовал знаковую работу, посвященную человеческому поведению, в которой была описана *иерархия потребностей*.[1] Маслоу классифицировал человеческие потребности в зависимости от их от-

[1] A. H. Maslow, "A Theory of Human Motivation," Psychological Review 50, no. 4 (1943): 370–396.

носительной важности и предположил, что после того, как человек удовлетворяет потребность нижнего уровня, у него может возникнуть интерес к удовлетворению потребности более высокого уровня. Этот иерархический подход может использоваться для описания другой категории сложных сущностей — программных проектов. На рис. 6.1 иерархия потребностей программного проекта изображена в виде пирамиды.

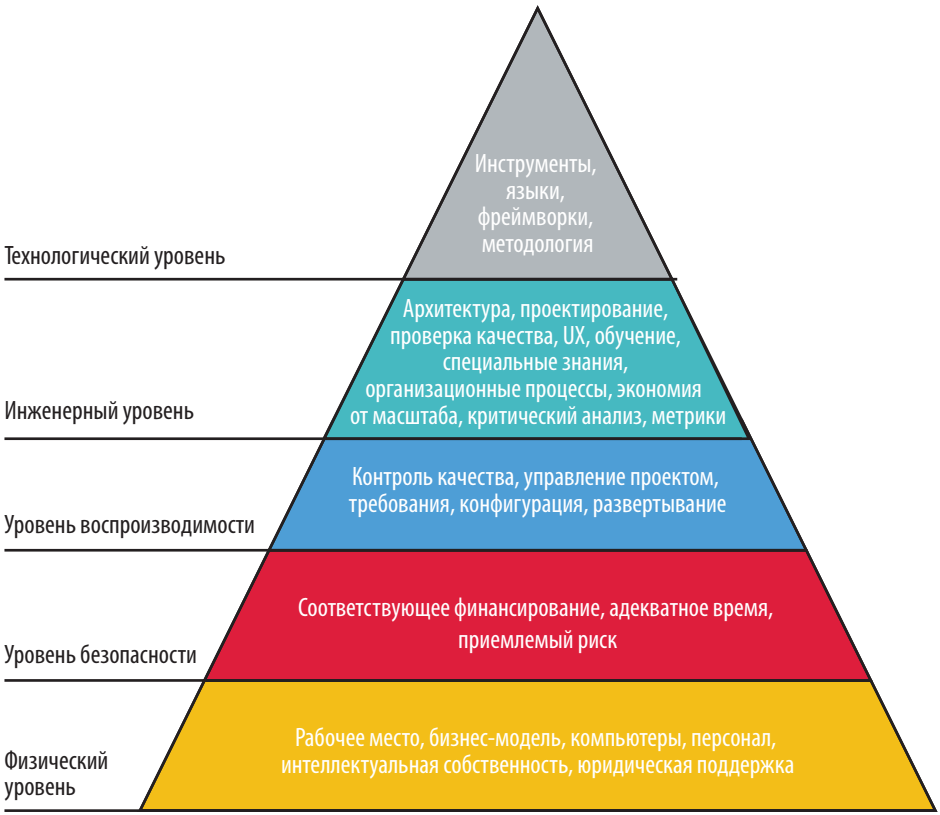


Рис. 6.1. Иерархия потребностей программного проекта

Потребности проекта можно разделить на пять уровней: физический, уровень безопасности, уровень воспроизводимости, инженерный и технологический уровни.

1. *Физический уровень*. Нижний уровень пирамиды, относящийся к физическому выживанию. Подобно тому как человеку необходимы воздух, вода, еда, одежда и убежище, проекту необходимо рабочее место (даже виртуальное) и жизнеспособная бизнес-модель. Проекту необходимы компьютеры для написания и тестирования кода, а также люди, назначенные для выпол-

нения этих задач. Проекту необходима правильная юридическая защита. Проект не должен нарушать существующую интеллектуальную собственность (IP), но при этом должен защищать собственную IP.

2. *Уровень безопасности*. После того как физические потребности будут удовлетворены, проект должен располагать адекватным финансированием (часто в форме выделенных ресурсов) и достаточным временем для завершения работы. Сама работа должна выполняться с приемлемым риском — не слишком безопасно (потому что проекты без риска обычно не стоят потраченного времени) и не слишком рискованно (потому что крайне рискованные проекты обычно завершаются неудачей). Короче говоря, проект должен быть безопасным до разумной степени. Планирование проекта работает на этом уровне пирамиды потребностей.
3. *Уровень воспроизводимости*. Воспроизводимость проекта описывает способность организации, занимающейся разработкой, успешно выдавать результаты раз за разом; она закладывает основу для управления и исполнения проекта. Воспроизводимость гарантирует, что если вы проводите планирование и берете на себя обязательства под определенный график и затраты, эти обязательства будут выполнены. Воспроизводимость отражает степень доверия к команде и проекту. Чтобы добиться воспроизводимости, необходимо управлять требованиями, контролировать ход проекта относительно плана, применять такие меры контроля качества, как модульное и системное тестирование, организовать эффективную систему управления конфигурацией, а также активно управлять развертыванием и операциями.
4. *Инженерный уровень*. После того как воспроизводимость проекта будет обеспечена, программный проект может впервые обратиться к таким аспектам разработки, как архитектура и проработка деталей, и таким операциям обеспечения качества, как анализ коренных причин и исправление неполадок (на общесистемном уровне), а также к профилактическим мерам. Первая часть этой книги, посвященная проектированию системы, работает на этом уровне пирамиды.
5. *Технологический уровень*. На этом уровне находятся технологии разработки, инструменты, методология, операционные системы и другие нетривиальные технические аспекты. Это вершина пирамиды потребностей, которая может полностью раскрыть свой потенциал только после того, как будут полностью удовлетворены все потребности более низких уровней.

В иерархии потребностей высокоуровневые потребности обслуживают интересы потребностей нижнего уровня. Например, по Маслоу, еда является потребностью более низкого уровня, чем работа, однако большинство людей работает для того, чтобы можно было есть (а не наоборот). Аналогичным образом технология обслуживает потребности инженерного уровня (например, архитектуры), а потребности инженерного уровня обслуживают потребности безопасности (предоставляемые планом проекта). Также это означает, что хро-

нологически необходимо начинать с проектирования системы; только после этого можно будет планировать проект для его построения.

Чтобы проверить структуру пирамиды, следует составить список всех необходимых составляющих успеха типичного программного проекта. После этого можно будет назначить потребностям приоритеты, отсортировать и, наконец, сгруппировать их по категориям.

В качестве эксперимента рассмотрим два проекта. Первый характеризуется архитектурой с сильными связями, высокой стоимостью сопровождения, низким уровнем повторного использования и сложностями с расширением. Тем не менее на выполнение работы выделено адекватное время, а проект нормально укомплектован персоналом. Во втором проекте используется великолепная архитектура: модульная, расширяемая и пригодная для повторного использования, удовлетворяющая все требования и готовая к возможным проблемам в будущем. Однако при этом команда недоукомплектована, но даже если бы необходимый персонал был доступен, на безопасную разработку системы не хватит времени. Спросите себя: в каком проекте я бы предпочел работать?

Ответ будет единодушным: в первом проекте. А следовательно, план проекта должен находиться в пирамиде потребностей на более низком уровне (то есть ближе к основанию), чем архитектура. Перевернутая пирамида потребностей — классическая причина неудач многих программных проектов. Представьте, что рис. 6.1 перевернут вверх ногами. Команда разработки почти полностью концентрируется на технологии, фреймворках, библиотеках и платформах; почти не выделяет время на архитектуру и проектирование; полностью игнорирует фундаментальные аспекты времени, затрат и риска. Пирамида потребностей становится нестабильной, поэтому вполне логично, что такие проекты завершаются неудачей. Вкладываясь в уровень безопасности пирамиды с использованием средств планирования проектов, вы структурируете потребности проекта по уровням, закладываете фундамент, который стабилизирует более высокие уровни, и ведете проект к успеху.

7 Обзор планирования проекта

В следующем обзоре описана базовая методология и приемы, используемые при планировании проекта. Хороший план проекта включает план подбора персонала, оценки объема работ и масштаба, план построения и интеграции сервисов, подробный график, калькуляцию затрат, жизнеспособность и подтверждение плана, а также организацию выполнения и отслеживания.

В этой главе описаны многие концепции планирования проекта, хотя некоторые подробности и одна-две критические концепции останутся для последующих глав. Тем не менее хотя материал этой главы задуман как простой обзор, в нем перечислены все основные составляющие успеха при проектировании и выпуске программных проектов. Также в ней приводятся обоснования некоторых фаз планирования с точки зрения разработки, тогда как в остальных главах приводится более техническая информация.

Определение успеха

Прежде чем читать дальше, необходимо понять, что суть планирования проекта — это успех и то, что требуется для его достижения. Отрасль разработки в целом показывала настолько неубедительные результаты, что она изменила само определение успеха: сегодня успехом считается все, что не приводит к немедленному банкротству компании. При такой низкой планке приемлемым считается практически все, от низкого качества до обмана в числах и недовольных заказчиков. Мое определение успеха отличается от этого, хотя оно тоже в каком-то смысле снижает планку требований. Я определяю **успех** как выполнение ваших обязательств.

Если вы запросили на проект один год и 1 миллион долларов, я ожидаю, что проект будет выполнен за один год, а не за два, и затраты на него составят 1 миллион, а не 3 миллиона. В отрасли разработки многим людям не хватает квалификации и знаний, необходимых для выполнения даже этого заниженно-

го критерия успеха. Идеи, представленные в этой главе, направлены исключительно на достижение этой цели.

Более высокое требование — выполнение проекта самым быстрым, наименее затратным и наиболее безопасным способом. Для реализации этой амбициозной цели потребуются приемы, описанные в следующих главах. Планку можно поднять еще выше и потребовать, чтобы архитектура системы оставалась качественной в течение десятилетий, чтобы она обеспечивала простоту сопровождения, возможности повторного использования и безопасность на протяжении всей своей долгой и благополучной жизни. Для этого вам неизбежно потребуются принципы проектирования, описанные в части I книги. Так как обычно вы сначала учитесь ходить и уже потом — бегать, лучше начать с базового уровня успеха и постепенно подниматься выше.

Как сообщать об успехе

В части I этой книги было сформулировано универсальное правило проектирования: *функции всегда и повсюду являются аспектами интеграции, а не реализации*. Как следствие, ни в каких ранних сервисах никаких функций нет. В какой-то момент интеграция сервисов достигнет уровня, при котором начнут просматриваться функции. Я называю эту точку **системой**. Система вряд ли появится в самом конце проекта, потому что на этой стадии могут выполняться дополнительные активности, такие как тестирование и развертывание системы. Как правило, система появляется ближе к концу, потому что для нее требуется большинство сервисов, а также клиентов. При использовании Метода это означает, что только после интеграции *Менеджеров*, *Ядер*, *Доступа к ресурсам* и *Вспомогательных средств* вы сможете поддерживать все поведение, необходимое для *Клиентов*.

И хотя система является продуктом интеграции, не вся интеграция происходит внутри *Менеджеров*. Часть интеграции происходит до завершения *Менеджеров* (например, интеграция *Ядер* с *Доступом к ресурсам*), а другая часть происходит уже после *Менеджеров* (например, между *Клиентами* и *Менеджерами*). Также может присутствовать явная деятельность по интеграции, например разработка клиента сервиса на модели с последующей интеграцией клиента с реальным сервисом.

Проблема с появлением системы на поздней стадии проекта — отрицательная реакция руководства. Многие люди, которым поручено управление разработкой программного продукта, не понимают концепций проектирования, описанных в книге, и просто требуют от вас реализации функций. Они ни на секунду не задумываются о том, что если работающая функция может появиться быстро и на ранней стадии, она не представляет особой ценности для бизнеса или заказчиков, потому что компания или команда не потратила на нее значительных усилий. Обычно руководство использует функции как метрику для

оценки успешности работы и нередко отменяет проблемные проекты со слабой динамикой. Проект сталкивается с серьезным риском: он может идти в идеальном соответствии с графиком, но поскольку система появляется только в конце, а отчеты от ходе проекта базируются на функциях, возникает угроза отмены. Проблема решается просто:

> *Отчет о ходе работы никогда не должен базироваться на функциях. Он всегда должен базироваться на интеграции.*

Во время работы над проектом, использующим Метод, выполняется множество интеграций. Такие интеграции малы и легко осуществимы. В результате проект начинает генерировать постоянный поток хороших новостей, что способствует формированию доверия и предотвратит необоснованную отмену проекта.

> **ПРИМЕЧАНИЕ** Такой подход к отчетности о ходе работ является стандартной практикой в других отраслях. Например, подрядчик, который строит дом, показывает домовладельцу фундамент, стены, закрепленные на фундаменте, подключение к коммунальным сетям и т. д. для того, чтобы развеять сомнения и опасения по поводу проекта, а не потому, что домовладельца интересуют все эти подробности.

Исходный подбор персонала для проекта

Хорошая архитектура не возникает сама по себе и не вырастает естественным образом при разумных затратах времени и средств. Хорошая архитектура — результат сознательных усилий со стороны архитектора. Как следствие, первым разумным действием в любом программном проекте становится назначение квалифицированного и компетентного архитектора. Чего-то меньшего будет недостаточно, потому что главным риском для любого проекта является отсутствие архитектора, ответственного за архитектуру. Этот риск затмевает любые другие исходные риски, с которыми может столкнуться проект. Неважно, насколько технически грамотны разработчики, насколько проверена технология и насколько идеально выстроена среда разработки. Ни один из этих факторов ни на что не повлияет, если система была спроектирована с дефектами. Воспользуемся аналогией с домом: хотели бы вы жить в доме, построенном из лучшего материала лучшей строительной бригадой в лучшем месте, но без какой-либо архитектуры (или с дефектной архитектурой)?

Архитектор, а не архитекторы

Архитектор должен выделить время на сбор и анализ требований, выявление базовых сценариев использования и областей нестабильности, проектирование системы и построение плана проекта. Само проектирование не занимает

много времени (архитектор обычно может спроектировать систему и создать план проекта за неделю-две), но на переход к точке, в которой архитектор может это сделать, иногда уходит несколько месяцев.

Многие руководители плохо реагируют как на проектирование в течение трехчетырех месяцев, так и на полный отказ от проектирования. Возможно, они захотят ускорить работу по проектированию и включить в нее еще нескольких архитекторов. Тем не менее анализ требований и проработка архитектуры занимают много времени и требуют напряженных размышлений. Назначение дополнительных архитекторов на эту работу нисколько не ускорит ее, а только усугубит ситуацию. Архитекторы обычно люди опытные и уверенные в своих силах, привыкшие работать независимо от других. Если назначить на проект нескольких архитекторов, они начнут конкурировать друг с другом, вместо того чтобы заниматься проектом.

Один из способов разрешения конфликтов между несколькими архитекторами — назначение комитета по проектированию. К сожалению, самый верный способ загубить любое начинание — назначить комитет для контроля этого начинания. Также можно разделить систему и поручить каждому архитектору конкретную область для проектирования. В этом случае система будет напоминать Химеру — чудовище из древнегреческой мифологии с головой льва, крыльями дракона, туловищем козы и хвостом змеи. Хотя каждая часть Химеры хорошо спроектирована и даже оптимизирована, Химера ни с чем не справляется хорошо: она не умеет летать, как дракон, бегать с быстротой льва или взбираться на скалы, как коза. Химере недостает целостности проектирования — и точно такая же ситуация возникает при проектировании системы несколькими архитекторами, каждый из которых отвечает за свою часть.

Один архитектор абсолютно необходим для целостного проектирования. Это наблюдение можно расширить до общего правила: единственный способ обеспечить целостность проектирования — поручить проектирование одному архитектору. Также справедливо и обратное: если в проекте нет специалиста, отвечающего за проектирование и способного наглядно представить его от начала до конца, такая система будет лишена целостности проектирования.

Кроме того, при нескольких архитекторах никто не отвечает за промежуточные версии, проектирование связей между подсистемами и даже между сервисами. В результате никто не отвечает за архитектуру системы в целом. А если за что-то никто не отвечает, это «что-то» никогда не будет сделано или, в лучшем случае, будет сделано плохо.

Если работа поручена одному архитектору, этот архитектор отвечает за проектирование системы. В конечном счете ответственность — единственный способ заслужить уважение и доверие руководства. Уважение всегда происходит от ответственности. Если единственного ответственного нет, как в случае с группой архитекторов, руководство на интуитивном уровне не испытывает к архитекторам и их работе ничего, кроме пренебрежения.

> **ВНИМАНИЕ** Наличие одного ответственного архитектора не означает, что работа архитектора не может анализироваться другими профессиональными архитекторами. Ответственность за проектирование не подразумевает работы в полной изоляции или запрета конструктивной критики. Наоборот, архитектор должен желать такого рецензирования, чтобы убедиться в адекватности своего решения.

Младшие архитекторы

В большинстве программных проектов достаточно одного архитектора. Этот принцип справедлив вне зависимости от размера проекта и крайне важен для успеха. Однако большие проекты могут очень легко опутать архитектора различными обязанностями, которые помешают ему сосредоточиться на ключевой цели — проектировании системы и борьбе с нарушениями архитектуры в ходе разработки. Кроме того, роль архитектора подразумевает техническое руководство, анализ требований, анализ архитектуры, анализ кода каждого сервиса в системе, обновление проектировочной документации, обсуждение запросов на реализацию функций от маркетинга и т. д.

Руководство может решить проблему лишней нагрузки, назначив на проект младшего архитектора (и даже нескольких младших архитекторов). Архитектор поручает многие вторичные задачи младшему архитектору, чтобы сосредоточиться на проектировании системы и планировании проекта в начале и защите архитектуры на протяжении работы над проектом. Конкуренция между архитектором и младшим архитектором маловероятна, потому что руководящая роль не вызывает сомнений, а границы ответственности четко обозначены. Наличие младших архитекторов также становится отличным способом обучения и приобретения опыта следующим поколением архитекторов в организации.

Основная команда

Какой бы важной ни была роль архитектора в проекте, он не может работать в изоляции. В день 1 у проекта должна быть сформирована основная команда. Она состоит из трех ролей: менеджер проекта, менеджер продукта и архитектор. Это логические роли, которые могут соответствовать или не соответствовать трем конкретным специалистам. Если соответствия нет, один человек может выполнять функции как архитектора, так и менеджера проекта, или проект может иметь нескольких менеджеров продукта.

В большинстве организаций и команд эти роли присутствуют, но названия конкретных должностей могут быть другими. Я определяю эти роли следующим образом:

- *Менеджер проекта.* Задача менеджера проекта — оградить команду от организации. Многие организации, даже небольшие, создают слишком много

шума. Если этот шум дойдет до команды разработки, он может парализовать ее работу. Хороший менеджер проекта работает как сетевой экран: он блокирует шум и пропускает только разрешенную информацию. Менеджер проекта следит за ходом работы и передает информацию о текущем состоянии высшему руководству и другим руководителям проекта, обсуждает условия и преодолевает общеорганизационные ограничения. Внутри компании менеджер проекта распределяет работу между разработчиками, планирует текущую деятельность и удерживает проект в рамках графика, бюджета и качества. Никто в организации, кроме менеджера проекта, не должен назначать работу или запрашивать информацию о текущем состоянии у разработчиков.

- *Менеджер продукта*. Менеджер продукта должен инкапсулировать заказчиков. Заказчики также являются постоянным источником шума. Менеджер продукта играет роль посредника при общении с ними. Например, если архитектору потребуется уточнить требуемое поведение, он не должен бегать за заказчиками; вместо этого ответы должен предоставить менеджер продукта. Менеджер продукта также разрешает конфликты между заказчиками (часто выраженные в виде взаимоисключающих требований), обсуждает требования, определяет приоритеты и передает ожидания относительно того, что считается приемлемым и на каких условиях.

- *Архитектор*. Архитектор — технический руководитель, который управляет проектированием, общей организацией работы и технической стороной проекта. Архитектор не только проектирует систему, но также следит за ней в ходе разработки. Архитектор должен работать с менеджером продукта для создания проектировочного решения и с менеджером проекта — для создания плана проекта. Хотя сотрудничество с менеджером продукта и менеджером проекта играет крайне важную роль, ответственность по обоим направлениям лежит на архитекторе. Как организатор процесса разработки, архитектор должен проследить за тем, чтобы команда строила систему в инкрементном режиме, неизменно придерживаясь ориентации на качество при проектировании системы и планировании проекта. Как техническому руководителю, архитектору часто приходится выбирать лучший способ решения технических задач («что делать»), оставляя подробности («как делать») разработчикам. Это требует непрерывного практического повышения квалификации, обучения и критического анализа.

Пожалуй, самым явным упущением в этом определении основной команды являются разработчики. Разработчики (и тестировщики) — временные ресурсы, которые приходят и перемещаются между проектами; это очень важное обстоятельство, к которому мы еще вернемся в этой главе при обсуждении планирования и распределения ресурсов.

В отличие от разработчиков, основная команда остается на всем протяжении работы над проектом, потому что все три роли необходимы проекту от начала

до конца. Но то, чем эти роли занимаются в проекте, изменяется со временем. Например, менеджер проекта переходит от согласований с ключевыми участниками к поставке отчетов о текущем состоянии, а менеджер продукта — от сбора требований к проведению демонстраций. Архитектор переходит от проектирования системы и планирования проекта к текущему техническому и организационному управлению (например, критический анализ архитектуры и кода на уровне сервисов и разрешение технических конфликтов).

Основная миссия

Миссия основной группы в начальной фазе — планирование проекта. В частности, это подразумевает надежные ответы на вопросы типа «сколько времени это займет?» или «сколько это будет стоить?». Невозможно получить ответы на эти ключевые вопросы без планирования проекта, а для планирования проекта необходима архитектура. В этом отношении архитектор всего лишь служит средством для достижения цели: планирования проекта. Так как архитектор должен взаимодействовать с менеджером продукта по поводу архитектуры и с менеджером проекта — по поводу планирования проекта, в начале проекта основная команда просто необходима.

Нечеткая начальная стадия

Основная команда планирует проект в нечеткой начальной стадии (*fuzzy front end*), ведущей к разработке. *Нечеткая начальная стадия* — общий термин[1] для любых технических проектов, обозначающий самое начало проекта. Начальная стадия наступает тогда, когда у кого-то появляется общее представление о проекте, и завершается, когда разработчики переходят к построению. Начальная стадия часто длится намного дольше, чем кажется многим людям: к тому времени, когда они начинают участвовать в проекте, начальная стадия может длиться уже несколько лет. Проекты сильно отличаются друг от друга, что ведет к нечеткости точной продолжительности начальной стадии. Продолжительность начальной стадии в наибольшей степени зависит от ограничений, установленных для проекта. Чем сильнее ограничен проект, тем меньше времени вы потратите в его начальной стадии. И наоборот, чем меньше ограничений, тем больше времени тратится на прогнозирование того, что вас ожидает в будущем и что с этим делать.

Программные проекты никогда не бывают полностью неограниченными. Любой проект сталкивается с некоторыми ограничениями по времени, масштабу, объему работы, ресурсам, технологии, унаследованному коду, бизнес-контексту и т. д. Такие ограничения могут быть как явными, так и неявными. Очень важно выделить время как на проверку явных ограничений, так и на выявление ограничений неявных. Проектирование системы и планирование проекта, на-

[1] https://en.wikipedia.org/wiki/Front_end_innovation

рушающие ограничение, — верный путь к катастрофе. По собственному опыту могу сказать, что в начальной стадии должно проходить от 15 до 25% всей продолжительности программного проекта (в зависимости от ограничений).

Обоснованные решения

Бессмысленно утверждать проект, не зная его истинных сроков, затрат и рисков. В конце концов, вы же не станете покупать дом, не зная заранее его цену. Вы не станете покупать дом, цена которого вас устраивает, но затраты на содержание и налоги окажутся неприемлемыми. Очевидно, брать обязательства по времени и средствам можно только после того, как станет известен масштаб. Многие программные проекты безрассудно продолжают действовать, не имея никакого представления о требуемом реальном времени и затратах.

Так же бессмысленно выделять ресурсы на проект до того, как организация согласится на проект и вы получите необходимое время и деньги. Выделение ресурсов на проект до согласия на него обычно продвигает проект вперед независимо от его целесообразности. Если самое правильное, что можно сделать, — это отказаться от реализации проекта, то организация только потеряет деньги. Спешка с выделением ресурсов почти всегда сопровождается плохим функциональным проектированием и полным отсутствием планирования — вряд ли это станет составляющими успеха.

Ключ к успеху заключается в принятии обоснованных решений, базирующихся на серьезном проектировании и оценках масштаба. Необоснованные ожидания — не стратегия, а интуиция — не знание, особенно когда вы имеете дело со сложными программными системами.

> **ПРИМЕЧАНИЕ** Неспособность принятия обоснованных решений по времени и затратам является постоянным источником разочарований для ключевых участников со стороны бизнеса при работе с командами программистов. Ответственные бизнесмены просто хотят знать необходимые затраты и то, в какой момент они смогут извлечь пользу из проведенных работ. Если вы будете обходить эти вопросы, это в конечном итоге создаст напряжение, подозрение и враждебность между командой и руководством. Люди со стороны бизнеса привыкли к планированию и распределению бюджетов. Они ожидают от профессиональных программистов, что те обладают необходимой квалификацией, чтобы действовать так же.

Планы, а не план

Результатом планирования проекта становится набор планов, а не отдельный план. Как упоминалось в предыдущей главе, план проекта не определяет единственную точку по времени и затратам. Всегда существует много воз-

можных вариантов построения любой системы, и только один вариант обеспечивает правильную комбинацию времени, затрат и риска. У архитектора может появиться искушение просто запросить у руководства параметры планирования проекта, а потом планировать этот единственный вариант. Проблема в том, что руководители не всегда говорят то, что думают (или думают не то, что говорят).

Для примера возьмем проект на 10 человеко-лет, то есть проект, в котором суммарные трудозатраты по всем активностям составят 10 человеко-лет. Допустим, руководство требует найти наименее затратный способ построения системы. Такой проект может быть реализован одним человеком за 10 лет, но руководство вряд ли захочет ждать 10 лет. Теперь предположим, что от вас требуют самый быстрый способ построения системы, и ту же систему можно построить, наняв 3650 человек на 1 день (или хотя бы 365 человек на 10 дней). Руководство вряд ли захочет нанимать столько людей на такие короткие сроки. Аналогичным образом руководство никогда не потребует от вас делать все максимально безопасно (потому что все, что вообще стоит строить, сопряжено с риском) или сознательно идти на самый рискованный вариант реализации проекта.

Критический анализ плана разработки программного продукта

Разрешить неоднозначность относительно того, чего на самом деле хочет руководство, можно только одним способом: предложить подборку хороших вариантов, каждый из которых представляет разумную комбинацию времени, затрат и риска. Эти варианты представляются руководству на специальной встрече, которая неофициально называется «кормить/убить». Как подсказывает название, на этой встрече руководство должно выбрать один из вариантов плана проекта и выделить необходимые ресурсы (путь «кормить»). Один из вариантов, который доступен всегда, — отказ от реализации проекта (путь «убить»). Официально такая встреча называется «анализом плана разработки продукта», или SDP (Software Development Plan Review). Неважно, если в вашем процессе разработки отсутствует точка для проведения SDP: просто организуйте встречу сами (ни один руководитель не откажет в просьбе о проведении встречи с темой «Анализ плана разработки продукта»).

После того как нужный вариант будет выбран, руководство должно буквально подписаться под документом SDP. Этот документ теперь становится страховым полисом вашего проекта: если вы не отклоняетесь от параметров плана, нет причин для отмены вашего проекта. Это требует правильной организации контроля (как описано в приложении А) и управления проектом.

Если ни один вариант не был признан приемлемым, вы должны привести руководство к правильному решению — в данном случае закрытию проекта.

Обреченный проект — тот, который с момента своего принятия не получил адекватного времени и ресурсов, — не принесет никакой пользы. В конечном итоге у проекта кончатся деньги и/или время, и организация не только потеряет средства и время, но и понесет издержки упущенной выгоды от направления этих ресурсов на другой жизнеспособный проект. Проект, у которого не было ни единого шанса на успех, также отрицательно скажется на карьере участников основной команды. Так как у вас всего несколько лет, чтобы завоевать репутацию и двигаться дальше, каждый проект должен идти в зачет и стать своего рода знаком отличия. Год или два, проведенные на второстепенном проекте, который завершился неудачей, ограничит ваши карьерные возможности. Закрытие такого проекта до начала разработки принесет пользу всем участникам.

Сервисы и разработчики

Располагая планом проекта (но только после того, как руководство выбрало конкретный вариант), команда может браться за построение системы. Обычно это требует распределения сервисов (или модулей, компонентов, классов и т. д.) между разработчиками. Конкретная методология назначения заслуживает собственного раздела позднее в этой главе. А пока примите к сведению, что сервисы всегда должны назначаться разработчикам в отношении 1 : 1. Отношение 1 : 1 не означает, что разработчик занимается только одним сервисом; скорее, если взять поперечное сечение команды в любой момент времени, вы увидите, что разработчик трудится над одним (и только одним) сервисом. Абсолютно нормально, если разработчик завершит один сервис и возьмется за следующий. Однако вы никогда не должны видеть разработчика, трудящегося над несколькими сервисами, или нескольких разработчиков, совместно работающих над одним сервисом. Любой другой способ распределения сервисов между разработчиками приведет к неудаче. Примеры неудачных вариантов назначения:

- **Несколько разработчиков на один сервис**. Назначение двух (и более) разработчиков на один сервис объясняется не избытком разработчиков, а желанием быстрее завершить работу. Тем не менее два человека не могут одновременно трудиться над одной задачей, поэтому приходится использовать одну из подсхем:
  - *Последовательная работа*. Разработчики могут работать поочередно, чтобы в любой момент над сервисом работал только один из них. Такая работа занимает больше времени из-за необходимости переключения контекста, то есть необходимости определять, что произошло с сервисом с того момента, когда текущий разработчик видел его в последний раз. Это противоречит исходной цели назначения двух разработчиков.

- *Параллелизм*. Разработчики могут работать параллельно, а затем интегрировать свою работу. Такая схема займет намного больше времени, чем работа одного разработчика над сервисом. Допустим, сервис, оцениваемый на один человеко-месяц, поручен двум разработчикам для параллельной работы. Появляется соблазн предположить, что работа будет выполнена через две недели, но это предположение ошибочно. Во-первых, не все единицы работы можно разделить таким образом. Во-вторых, разработчикам придется выделить по крайней мере еще одну неделю на интеграцию своей работы. Успех интеграции совершенно не гарантирован, если разработчики работали параллельно и не сотрудничали в ходе разработки. Даже если интеграция возможна, она обнулит всю работу по тестированию в каждой части из-за изменений интеграции. Тестирование сервиса в целом потребует дополнительного времени. В целом работа займет не менее месяца (а скорее всего, больше). При этом другие разработчики, работающие над зависимыми сервисами и ожидающие, что сервис будет готов через две недели, столкнутся с дополнительной задержкой.

- **Несколько сервисов на одного разработчика**. Вариант с назначением двух (и более) сервисов одному разработчику тоже плох. Предположим, два сервиса, A и B, каждый из которых, по оценке, потребует месяца работы, назначены одному разработчику. Разработчик должен закончить оба сервиса через месяц. Так как суммарный объем работы составляет два месяца, работа не только не будет закончена за месяц, но и потребует намного больше времени. Пока разработчик трудится над сервисом A, он не может работать над сервисом B. Разработчики, зависящие от сервиса B, начинают требовать, чтобы разработчик занялся сервисом B. Разработчик может переключиться на сервис B, но тогда внимания потребуют другие разработчики, зависящие от сервиса A. Все эти переключения серьезно снижают эффективность труда разработчика, а работа затягивается намного более чем на два месяца. В итоге сервисы A и B будут готовы через три или четыре месяца.

Назначение нескольких разработчиков на один сервис или нескольких сервисов на одного разработчика приводит к тому, что по проекту распространяется каскад задержек, в основном обусловленных зависимостями, влияющими на других разработчиков. В свою очередь, это сильно затрудняет точные оценки. Единственный вариант, который обеспечивает хоть какую-то ответственность и вероятность выполнения оценок, — распределение сервисов между разработчиками в отношении 1 : 1.

> **ПРИМЕЧАНИЕ** Распределение сервисов между разработчиками не исключает парного программирования. И хотя парное программирование удваивает количество разработчиков, назначенных одновременно на один сервис, эта пара не будет работать последовательно или параллельно.

Проектирование и эффективность команды

При распределении сервисов между разработчиками 1 : 1 следует, что взаимодействие между сервисами изоморфно взаимодействию между разработчиками. Посмотрите на рис. 7.1.

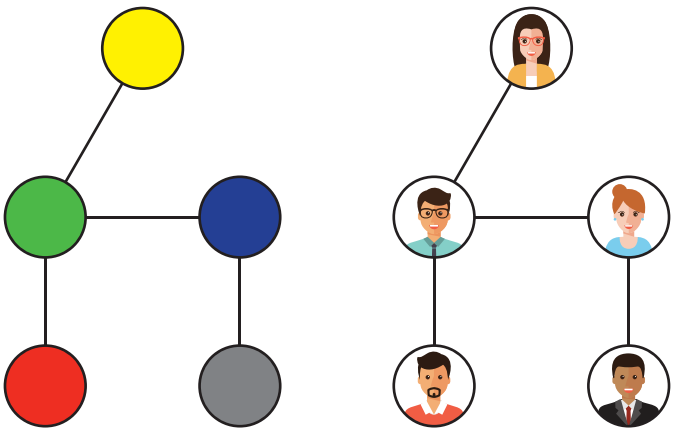


Рис. 7.1. Архитектура системы — архитектура команды (изображения: Sapann Design/Shutterstock)

Отношения между сервисами, их взаимодействия и коммуникации диктуют отношения и взаимодействия между разработчиками. При распределении 1 : 1 архитектура системы определяет архитектуру команды.

Хорошая архитектура системы стремится к сокращению количества взаимодействий между модулями до минимума — полная противоположность тому, что происходит на рис. 7.2. Система со слабыми связями (вроде изображенной на рис. 7.1) минимизирует количество взаимодействий до уровня, при котором с удалением одного взаимодействия система становится неработоспособной.

Архитектура на рис. 7.2 явно обладает сильной связанностью; кроме того, она описывает механизм работы команды. Сравните команды на рис. 7.1 и 7.2. К какой команде вы бы предпочли присоединиться? Команда на рис. 7.2 — стрессовая и непрочная. Скорее всего, участники команды будут сопротивляться изменениям, потому что каждое изменение будет создавать каскадные эффекты, которые разрушают их работу и работу других участников. Им придется проводить запредельное время на собраниях для решения возникших проблем. Напротив, команда на рис. 7.1 может решать проблемы локально и ограничивать их последствия. Каждый участник команды практически независим от остальных, ему не приходится тратить много времени на координацию работы. Проще говоря, команда на рис. 7.1 намного эффективнее команды на рис. 7.2.

В результате команда с лучшей архитектурой системы имеет гораздо лучшие перспективы соблюдения жестких сроков.

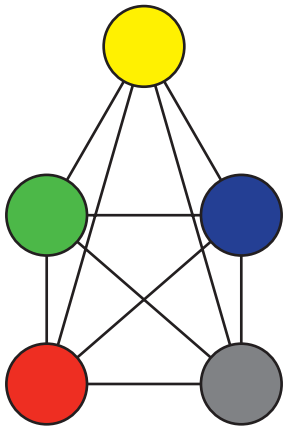


Рис. 7.2. Сильные связи в системе и команде

Последнее наблюдение исключительно важно: многие руководители признают полезность системного проектирования только на словах, потому что преимущества архитектуры (простота сопровождения, расширяемость и возможность повторного использования) проявляются лишь в перспективе. Будущие преимущества никак не помогут руководителю, который сталкивается с суровой реальностью нехватки ресурсов и жесткого графика. Раз на то пошло, оно позволит руководителю максимально сократить объем работы для соблюдения срока. Так как проектирование системы вроде бы не помогает с достижением текущих целей, руководитель выкинет за борт все осмысленные затраты на проектирование. К сожалению, при этом он теряет любую возможность соблюдения обязательств, потому что соблюдение жестких сроков возможно только с первоклассным проектировочным решением, которое обеспечивает максимальную эффективность работы команды. Когда вы стремитесь получить поддержку руководства для своей работы по проектированию, покажите, как проектирование помогает с достижением непосредственных целей, — а уже из этого последуют долгосрочные преимущества.

Личные отношения и проектирование

Влияние архитектуры на эффективность команды может быть очевидным, но команда тоже влияет на архитектуру. Если на рис. 7.1 два разработчика не общаются друг с другом, эта область архитектуры будет слабой. Два взаимосвязанных сервиса следует поручать двум разработчикам, которые способны к эффективному взаимодействию.

Непрерывность задач

При назначении сервисов (или таких активностей, как разработка пользовательского интерфейса) старайтесь поддерживать *непрерывность задач* — логическое продолжение задач, поручаемых каждому участнику. Часто такие назначения задач соответствуют графу зависимости сервисов. Если сервис A зависит от сервиса B, поручите A разработчику B. Одно из преимуществ такого подхода — если разработчик A уже знаком с B, ему потребуется меньше времени на то, чтобы войти в курс дела. Еще одно важное преимущество непрерывности задач, о котором часто забывают, — совмещение критериев успеха разработчика и проекта. У разработчика появляется стимул хорошо выполнить работу B, чтобы избежать проблем, когда придет время заняться A. Идеальная непрерывность задач вряд ли достижима, но к ней следует стремиться.

Наконец, учитывайте личные технические предпочтения разработчиков при назначении задач. Например, эксперту по безопасности с большой вероятностью не стоит поручать построение пользовательского интерфейса, эксперту по базам данных — реализацию бизнес-логики, а джуниорам — реализацию таких подсистем, как шина сообщений или диагностика.

Оценка трудозатрат

Оценка затрат рабочего времени, или оценка трудозатрат, используется для получения ответа на вопрос: сколько времени займет выполнение некоторой задачи? Существуют два вида таких оценок: оценки отдельных активностей (оценка трудозатрат для активности, закрепленной за ресурсом) и общая оценка для проекта. Два типа оценок не связаны друг с другом, потому что общая продолжительность проекта не равна сумме оценок трудозатрат по всем активностям, разделенной на количество ресурсов. Это связано с внутренней неэффективностью использования людских ресурсов, внутренними зависимостями между активностями, а также всеми мерами по сокращению рисков, которые вам придется применить.

Во многих командах участие в формировании оценок становится в лучшем случае симпатичным ритуалом, а в худшем — пустой тратой времени. Плохие результаты оценок в отрасли разработки объясняются несколькими причинами:

1. Неопределенность того, сколько времени займет активность, и даже неопределенность в списке активностей — главная причина плохой точности оценок. Не путайте причину и следствие: неопределенность — причина, а плохая точность оценок — результат. Вы должны сознательно сокращать неопределенность, как описано позднее в этой главе.
2. Мало кто из специалистов по разработке ПО обучен простым и эффективным методам оценки. Большинству приходится пользоваться догадками, интуицией и предвзятым мнением.

3. Многие люди склонны завышать или занижать оценки в попытке компенсировать неопределенность, что значительно ухудшает результаты.
4. При составлении списка активностей нередко учитывается только верхушка айсберга. Естественно, если вы упустите активности, необходимые для успеха, ваши оценки будут смещены. Это справедливо как для активностей в рамках проекта, так и для внутренних фаз между активностями. Например, оценщики могут ограничиться только активностями из области программирования или в рамках активностей программирования учесть написание кода, но не проектирование или тестирование.

Классические ошибки

Как упоминалось выше, люди склонны завышать или занижать оценки в попытке компенсировать неопределенность. Оба отклонения фатальны для успеха проекта.

Завышенные оценки никогда не работают из-за закона Паркинсона.[1] Например, если вы выделите разработчику три недели на выполнение двухнедельной операции, разработчик не будет работать две недели, чтобы потом неделю простаивать. Вместо этого разработчик будет работать над задачей все три недели. Так как фактическая работа займет только две недели из трех, в дополнительную неделю разработчик будет заниматься «украшательством» — добавлением всевозможных излишеств, аспектов и возможностей, которые никому не нужны и не были частью исходной архитектуры. Украшательство значительно повышает сложность задачи, а повышение сложности радикально снижает вероятность успеха. Из-за этого разработчику требуется четыре-шесть недель для завершения исходной задачи. Другие разработчики в проекте, ожидающие получить код через три недели, тоже задерживаются. Более того, команде достается (возможно, на многие годы и на несколько ближайших версий) программный модуль, который без всякой необходимости оказывается более сложным, чем предполагалось изначально.

Заниженная оценка также становится гарантией неудачи. Бесспорно, если выделить разработчику два дня на задачу, требующую двухнедельного программирования, это предотвратит любое украшательство. Проблема в том, что разработчик будет стараться выполнить свою работу на скорую руку, срезая углы и игнорируя все известные общепринятые практики. Такой подход ничуть не разумнее, чем проведение на скорую руку хирургической операции или строительства дома.

К сожалению, никакую нетривиальную задачу невозможно выполнить на скорую руку. Так как разработчик отказывается от всех общепринятых практик в области разработки, от тестирования до подробного проектирования и напи-

[1] Cyril N. Parkinson, «Parkinson's Law», *The Economist* (November 19, 1955).

сания документации, разработчик теперь пытается решить задачу худшим из всех возможных способов. Соответственно, разработчик не будет работать над задачей номинальные две недели, которые она должна была занять при условии правильного выполнения работы, а проведет за ней четыре-шесть недель (или больше) из-за низкого качества и возросшей сложности. Как и при завышении оценки, другие разработчики, ожидающие получить код через запланированные два дня, сталкиваются с задержкой. Наконец, команде достается (возможно, на многие годы и на несколько ближайших версий) программный модуль, который написан худшим из всех возможных способов.

Вероятность успеха

Хотя эти рассуждения выглядят разумно, многие разработчики упускают из виду величину этих классических ошибок. На рис. 7.3 изображен график вероятности успеха как функции оценки. Для примера возьмем однолетний проект. При правильной архитектуре и планировании проекта нормальная оценка для проекта составляет 1 год (точка *N*). А какой будет вероятность успеха, если выделить на проект один день? Неделю? Месяц? Очевидно, при достаточно жестких оценках вероятность успеха равна нулю. А как насчет 6 месяцев? Хотя вероятность завершения однолетнего проекта за 6 месяцев чрезвычайно низка, она отлична от нуля, потому что может произойти чудо. С другой стороны, вероятность успеха при оценке в 11 месяцев и 3 недели весьма высока, и она останется достаточно высокой при 11 месяцах. Тем не менее завершение проекта за 9 месяцев маловероятно. Следовательно, слева от нормальной оценки находится переломная точка, с которой вероятность успеха начинает резко расти в нелинейной зависимости. Аналогичным образом однолетний проект может занять 13 месяцев, и даже продолжительность 14 месяцев выглядит

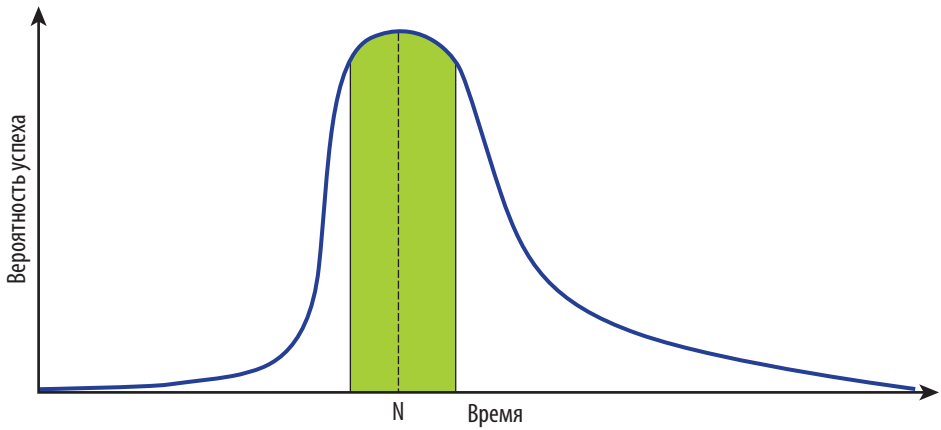


Рис. 7.3. Вероятность успеха как функция оценки (адаптировано по материалам Steve McConnell, Rapid Development (Microsoft Press, 1996))

разумно. Но если выделить на проект 18 или 24 месяца, это наверняка загубит его, потому что вступит в силу закон Паркинсона: работа заполняет все отпущенное на нее время, и проект ждет крах из-за возрастания сложности. Однако справа от нормальной оценки существует другая переломная точка, в которой вероятность успеха снова падает в нелинейной зависимости. На рис. 7.3 продемонстрировала огромная важность хороших номинальных оценок, потому что они обеспечивают максимальную вероятность успеха с нелинейными последствиями отклонений. Скорее всего, в прошлом вам уже доводилось навредить себе и другим из-за заниженных и завышенных оценок. Классические ошибки не просто часто встречаются — они являются фундаментальными.

Методы оценки

Плохие показатели эффективности оценок в программной отрасли сохраняются даже несмотря на то, что эффективные методы оценки, подходящие для разных отраслей, существуют уже несколько десятилетий. Я еще не видел ни одной команды, правильно подходившей к формированию оценок, которая бы при этом промахнулась с планированием проекта и обязательствами. Вместо того чтобы пытаться дать обзор всех методов такого рода, в этом разделе будут выделены идеи и приемы, которые, на мой взгляд, оказались достаточно простыми и эффективными за прошедшие годы.

Точность оценки, а не степень точности

Хорошие оценки точны, но не в числовом отношении. Например, представьте некую задачу, на которую потребовалось 13 дней. Для задачи были предложены две оценки: 10 дней и 23,8 дня. Хотя вторая оценка обладает большей числовой точностью, первая оценка безусловно лучше, потому что она ближе к фактическому значению. В оценках точность прогноза важнее числовой точности. Так как многие программные проекты значительно отклоняются от своих обязательств (иногда в разы по сравнению с исходными оценками), бессмысленно оценивать такие проекты с точностью до часа.

Оценки также должны соответствовать дискретности отслеживания. Если менеджер проекта отслеживает состояние проекта на еженедельной основе, любая оценка менее недели будет бессмысленной, потому что меньше дискретности измерений. Это так же бессмысленно, как оценка размера дома с точностью до микрона, когда вы измеряете его рулеткой.

Даже если активность занимает всего 13 дней, лучше оценить ее в 15 дней, чем в 12,5. Любой сколько-нибудь серьезный проект будет состоять из десятков активностей; стремясь к точности оценки, вы с большой вероятностью будете завышать (немного) оценки для одних активностей и занижать (немного) для других. Но в среднем ваши оценки будут довольно точными. Попытки дать оценку, точную в числовом отношении, могут привести к накоплению ошибок,

потому что ошибки в оценках не будут компенсироваться. Кроме того, если вы будете требовать у людей оценок с высокой числовой точностью, это только приведет к долгим спорам. Если же вы запросите адекватную оценку, то результат будет получен быстро и просто.

Сокращение неопределенности

Неопределенность — главная причина ошибочных оценок. Важно не путать неизвестное с неопределенным. Например, хотя точный день моей смерти неизвестен, его никак не назовешь неопределенным. На способности прогнозирования этой даты строится целая отрасль (страхование жизни). Хотя оценка может быть неточной в том, что касается меня лично, у страховщиков достаточно клиентов, чтобы результат был достаточно точным.

Обращаясь к людям за оценками, помогите им преодолеть страх оценок. У многих есть печальный опыт, когда их неточные оценки в прошлом использовались против них. Кто-то даже может отказаться от оценки со словами «Я не знаю» или «Оценки вообще не работают». Возможно, такие люди опасаются скрытой ловушки, или пытаются увильнуть от работы по оцениванию, или просто невежественны и неопытны в методах оценки — дело не в принципиальной неспособности оценивать.

Столкнувшись с неопределенностью, выполните следующие действия:

- Сначала предложите оценить хотя бы порядок величины: день, неделя, месяц, год? Определившись с порядком, постарайтесь сузить диапазон с множителем 2. Например, если на первый вопрос в качестве единицы измерения был выбран месяц, спросите, какой срок кажется более вероятным — две недели, месяц, два месяца, четыре месяца? Первый ответ исключает как восемь месяцев (потому что этот срок ближе к году), так и одну неделю, потому что она не была указана изначально как порядок величины.
- Постарайтесь составить список областей неопределенности в проекте и сосредоточьтесь на их оценке. Всегда разбивайте большие активности на меньшие, более удобные, чтобы значительно повысить точность оценок.
- Проведите подготовительную работу, которая даст представление о природе задачи и сократит неопределенность. Проанализируйте историю работы команды в организации и определите по ней, сколько времени занимали аналогичные задачи в прошлом.

Оценки PERT

Один из приемов оценки, предназначенный специально для условий высокой неопределенности, является частью методологии PERT (Program Evaluation and Review Technique[1]). Для каждой активности приводятся три оценки: самая

[1] https://ru.wikipedia.org/wiki/PERT

оптимистическая, самая пессимистическая и наиболее вероятная. Итоговая оценка вычисляется по следующей формуле:

$$E = \frac{O + 4 \times M + P}{6},$$

где:

- E — вычисленная оценка;
- O — оптимистическая оценка;
- M — наиболее вероятная оценка;
- P — пессимистическая оценка.

Например, если для активности предоставлена оптимистическая оценка 10 дней, пессимистическая оценка 90 дней и наиболее вероятная оценка 25 дней, то оценка PERT для нее составит 33,3 дня:

$$E = \frac{10 + 4 \times 25 + 90}{6} = 33{,}3.$$

Общая оценка проекта

Оценка проекта в целом полезна в первую очередь для проверки планирования архитектуры, но она также может пригодиться в начале планирования. Когда вы завершите подробный план проекта, сравните его с общей оценкой проекта. Две величины не обязаны точно совпадать, но они должны быть согласованными и подтверждающими друг друга. Например, если подробный план проекта рассчитан на 13 месяцев, а общая оценка проекта составила 11 месяцев, то подробный план проекта действителен. Но если общая оценка составила 18 месяцев, но по крайней мере одно из двух чисел ошибочно, вы должны исследовать причину расхождений. Также общая оценка проекта может пригодиться в проектах, имеющих очень мало предварительных ограничений. Такой проект — «чистый холст» — имеет слишком много неизвестных факторов, затрудняющих проектирование. Общая оценка может использоваться для обратной проработки с целью установления ограничений для некоторых активностей, упрощающих исходный процесс планирования проекта.

Исторические источники

Для общей оценки проекта самую важную роль играет ваш опыт выполнения работ и история. Даже при умеренной степени воспроизводимости (см. рис. 6.1) маловероятно, что вам удастся реализовать проект быстрее или медленнее, чем аналогичные проекты в прошлом. Определяющим фактором результативности является природа организации, ее собственный уровень зрелости, а эти

характеристики не изменятся за один вечер или между проектами. Если вашей компании понадобился год для реализации аналогичного проекта в прошлом, то и в будущем тоже потребуется год. Возможно, этот проект в другом месте кто-то реализовал бы за полгода, но в вашей компании он займет год. Впрочем, есть и хорошие новости: воспроизводимость также означает, что на завершение такого проекта вряд ли потребуется два или три года.

Средства оценки

Превосходный, хотя и малоизвестный метод общей оценки проекта — использование программных средств оценки проектов. Обычно такие программы предполагают некоторую нелинейную связь между размерами и затратами (например, степенную) и используют большой объем данных ранее проанализированных проектов для обучения. Некоторые средства даже используют моделирование методом Монте-Карло для сужения диапазона переменных на основании атрибутов проекта или исторических данных. Я пользовался такими программами уже много лет, и они выдают точные результаты.

Широкополосная оценка

Широкополосная оценка — моя версия широкополосного дельфийского метода оценки[1]. Широкополосная оценка использует множество отдельных оценок для определения среднего значения общей оценки проекта, а затем добавляет полосы оценок выше и ниже него. Оценки вне полосы используются для лучшего понимания природы проекта и уточнения оценок, а процесс повторяется до того, как две оценки сойдутся.

Чтобы начать работу по широкополосной оценке, сначала соберите большую группу ключевых участников проекта, от разработчиков до тестировщиков, руководителей и даже представителей службы поддержки — разнообразие группы играет ключевую роль в широкополосном методе. Постарайтесь собрать всех: новичков и ветеранов, «адвокатов дьявола», экспертов и универсалов, людей творческих и простых работяг. Вам нужно подключиться к синергии знаний, интеллекта, опыта, интуиции и оценок рисков этой группы. Хороший размер группы — от 12 до 30 человек. Группы с менее чем 12 участниками возможны, но статистический элемент может оказаться недостаточно сильным для получения хороших результатов. Если участников более 30, оценку будет трудно получить за время одной встречи.

Начните встречу с краткого описания текущего состояния и фазы проекта; того, что уже было сделано (например, архитектура); и дополнительной контекстной информации (например, оперативных концепций системы), которые могут быть неизвестны ключевым участникам, не входящим в основную

[1] Barry Boehm, *Software Engineering Economics* (Prentice Hall, 1981).

команду. Каждый участник должен предоставить две числовые оценки для проекта: сколько он займет в месяцах и сколько людей для него потребуется. Предложите участникам записать эти числа рядом со своим именем на листке бумаги. Соберите листки, введите их в электронную таблицу и вычислите среднее значение и стандартное отклонение для каждого значения. Теперь найдите оценки (по времени и персоналу), находящиеся на расстоянии по крайней мере одного стандартного отклонения от среднего, то есть оценки, выходящие за пределы широкой полосы консенсуса (отсюда и название метода). Эти оценки являются статистическими выбросами.

Вместо того чтобы исключать выбросы из анализа (стандартная практика в большинстве статистических методов), запросите информацию у их авторов — возможно, они знают что-то такое, что неизвестно другим. Это отличный способ поиска неопределенностей. После того как авторы выбросов приведут свои обоснования для оценки и все будут с ними ознакомлены, снова соберите оценки. Процесс повторяется до тех пор, пока все оценки не будут лежать в границах одного стандартного отклонения или отклонение будет меньше дискретизации измерений (например, одного человека или одного месяца). В широкополосном методе оценки обычно сходятся к третьему кругу оценивания.

> **ВНИМАНИЕ** В ходе встречи важно поддерживать товарищескую, благожелательную атмосферу. Авторы аномальных оценок (как высоких, так и низких) известны всем участникам процесса, и их оценки не должны восприниматься как критика управления или организации.

Предупреждение

Общая оценка проекта, построенная на основании исторических данных, средств оценки или широкополосного метода, обычно бывает точной (если не *очень* точной). Сравните разные общие оценки, чтобы убедиться в том, что ваша оценка получилась действительно хорошей. К сожалению, хотя эти общие оценки точны, они всего лишь дополняют и проверяют вашу работу по планированию проекта. Они только подкрепляют план и обеспечивают проверку на разумность, потому что сами по себе не дают оснований для немедленных действий. Вы можете быть вполне уверены в том, что проект займет 18 месяцев с 6 людьми, но при этом понятия не иметь, как использовать эти ресурсы для завершения проекта по графику. Чтобы получить эту информацию, необходимо заняться планированием проекта.

Оценки отдельных активностей

Планирование проекта начинается с оценки продолжительности отдельных активностей проекта. Прежде чем оценивать отдельные активности, необходимо составить подробный список всех активностей в проекте — как непосред-

ственно связанных с программированием, так и других. В некотором смысле даже список активностей является оценкой фактического набора активностей, поэтому в данном случае действуют те же обоснования относительно сокращения неопределенностей. Избегайте соблазна сосредоточиться на основополагающих активностях программирования, обозначенных системной архитектурой, и активно «заглядывайте под воду», чтобы получить представление о полных размерах айсберга. Выделите время на поиск активностей и предложите другим людям составить такой же список, чтобы сравнить его с вашим собственным списком. Пусть коллеги анализируют, критикуют и ставят под сомнение ваш список активностей. Вас удивит, сколько всего вы упустили.

Так как точность оценки важнее числовой точности, при любых оценках активностей желательно всегда использовать 5-дневный квант. Активности, занимающие 1–2 дня, не должны входить в план. Активности, занимающие 3 или 4 дня, всегда оцениваются в 5 дней. Активности могут занимать 5, 10, 15, 20, 25, 30 или 35 дней. Активности, оцениваемые на 40 и более дней, могут быть хорошими кандидатами для разбиения на меньшие активности ради сокращения неопределенности. Использование сроков, кратных 5 дням, обеспечивает удобное выравнивание проекта по неделям и уменьшает количество «обрезков» из неполных недель до и после активности. Такая практика также соответствует реальной жизни — активности никогда не начинаются по пятницам.

Сокращение неопределенности приносит пользу даже для активностей обычного размера. Заставляйте себя и других разбивать каждую активность на задачи, не связанные напрямую с кодированием, — например, период обучения, тестирование клиентов, установка, точки интеграции, партнерская проверка и документирование. Если вы не будете концентрироваться исключительно на программировании и заранее проанализируете полный объем работы, вы быстро сократите неопределенность оценок отдельных активностей.

Диалог при получении оценок

Если вы хотите, чтобы другие люди предоставляли свои оценки для активности, с ними необходимо правильно вести диалог. Никогда не диктуйте продолжительность фразами вроде «У тебя две недели!». Мало того что такая оценка ни на чем не основана — владелец активности также не чувствует себя ответственным за то, чтобы действительно завершить работу за две недели. Если люди не чувствуют себя ответственными, это отразится на скорости и качестве. Избегайте наводящих вопросов типа «Это займет пару недель, верно?». Хотя это немного лучше, чем напрямую продиктованная оценка, вы подталкиваете другую сторону к своей оценке. Даже если человек согласен с вами, он не будет чувствовать ответственность за вашу оценку. Намного лучше спросить напрямую: «Сколько времени займет эта работа?» Не соглашайтесь на немедленный ответ. Всегда заставляйте собеседника вернуться к вам позднее, потому что вы хотите, чтобы он проанализировал задействованные факторы, а также пораз-

мыслил над ответом. Хорошие оценки необходимы для достижения максимальной вероятности успеха и ответственности участников (рис. 7.3).

Анализ критического пути

Чтобы вычислить фактическую продолжительность проекта, а также ряд других ключевых аспектов проекта, необходимо найти *критический путь* проекта. Анализ критического пути — самый важный метод планирования проекта. Тем не менее этот анализ не может выполняться без следующих предварительных условий:

- *Архитектура системы.* У вас должна быть готова декомпозиция системы на сервисы и другие структурные элементы (например, *Клиенты* и *Менеджеры*). Хотя проект можно спланировать даже с плохой архитектурой, безусловно, такая ситуация далека от идеала. Плохая архитектура системы продолжит изменяться, а вместе с ней будет изменяться ваш план проекта. Очень важно, чтобы архитектура системы была проверена, чтобы она оставалась истинной с течением времени.
- *Список всех активностей проекта.* В список должны быть включены как активности, непосредственно связанные с программированием, так и другие. Список большинства активностей, связанных с программированием, достаточно просто строится на основе анализа архитектуры. Список остальных активностей строится так, как было описано ранее, и также зависит от природы бизнеса. Например, в компании, разрабатывающей программное обеспечение для банков, будут присутствовать активности, связанные с проверкой соответствия законодательству и правовым нормам.
- *Оценка объемов работ для активностей.* Постройте точную оценку объема работы для каждой активности в списке. Используйте несколько методов оценки для повышения точности.
- *Дерево зависимостей сервисов.* Используйте цепочки вызовов для выявления зависимостей между разными сервисами в архитектуре.
- *Зависимости между активностями.* Кроме зависимостей между сервисами, необходимо построить список влияния всех активностей на другие активности (как непосредственно связанные с программированием, так и все остальные). Добавляйте интеграционные активности по мере необходимости.
- *Параметры планирования.* Вы должны знать ресурсы, доступные для проекта, а точнее, сценарии комплектования, которых требует ваш план. Если таких сценариев несколько, вы должны иметь разные планы проекта для каждого доступного сценария. Параметры планирования также включают типы требуемых ресурсов для каждой фазы проекта.

Сетевой график проекта

Все активности проекта можно представить в графической форме в виде сетевого графика. На сетевом графике проекта изображены все активности проекта и зависимости между ними. Сначала зависимости между активностями выводятся из того, как цепочки вызовов распространяются в системе. Для каждого из проверенных сценариев использования должна быть представлена цепочка вызовов, или диаграмма последовательности, которая показывает, как некоторое взаимодействие между структурными элементами системы поддерживает каждый сценарий использования. Если на одной диаграмме *Клиент А* вызывает *Менеджер А*, А на второй диаграмме *Клиент А* вызывает *Менеджер В*, то *Клиент А* зависит как от *Менеджера А*, так и от *Менеджера В*. Таким образом вы систематически выявляете зависимости между компонентами архитектуры. На рис. 7.4 изображена диаграмма зависимостей между программными модулями в условной архитектуре, построенной на основе Метода.

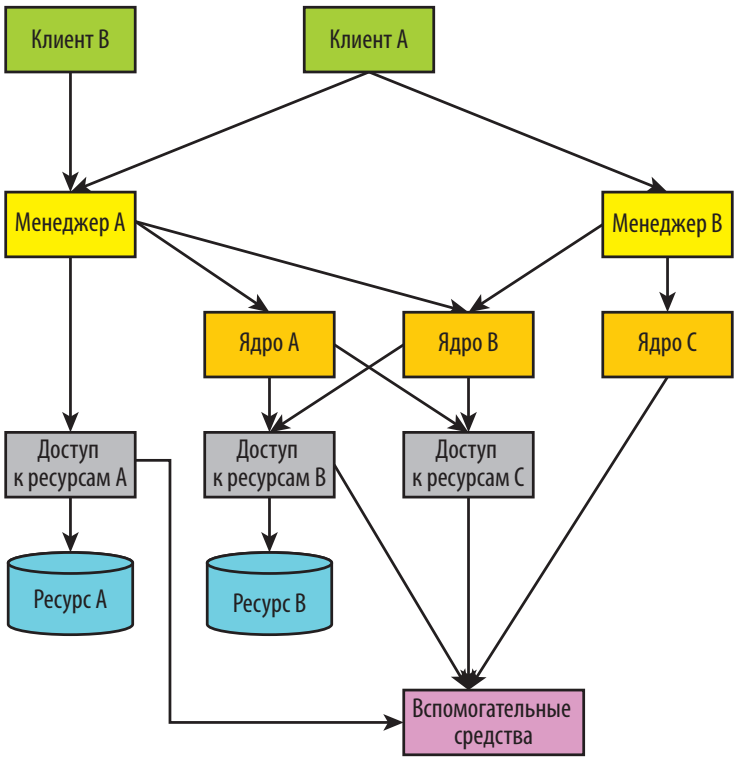


Рис. 7.4. Диаграмма зависимостей между сервисами

Диаграмма зависимостей на рис. 7.4 обладает рядом недостатков. Во-первых, она сильно структурирована и на ней отсутствуют все неструктурные активности (как связанные с программированием, так и все остальные). Во-вторых, она выглядит громоздко, а в более крупных проектах такие диаграммы получатся перегруженными и неудобными. В-третьих, группировка активностей (как в случае с категорией *Вспомогательные средства*) нежелательна.

Вы должны превратить диаграмму на рис. 7.4 в подробную абстрактную диаграмму, изображенную на рис. 7.5. Теперь диаграмма содержит все активности (как связанные с программированием, так и все остальные) — например, разработку архитектуры и тестирование системы. Возможно, на диаграмму также стоит добавить боковую панель с условными обозначениями, на которой перечислены активности для удобства просмотра.

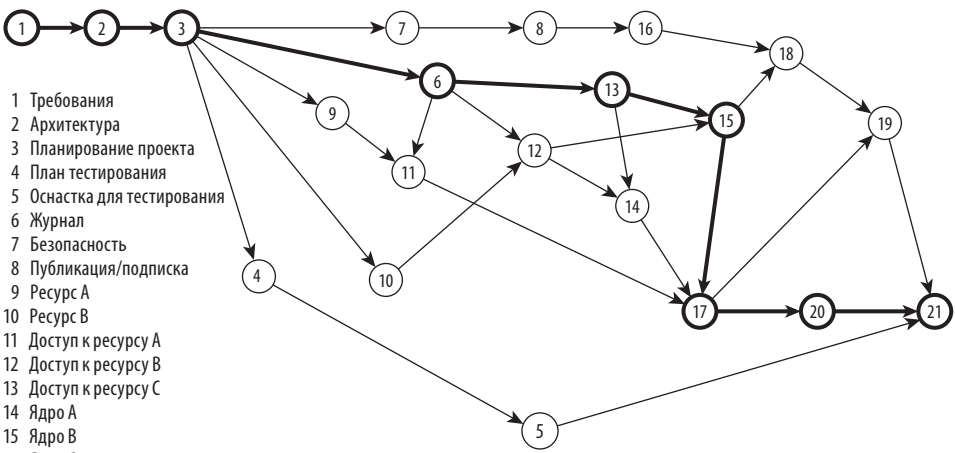


Рис. 7.5. Сетевой график проекта

Время активностей

Оценка объема работы для активности сама по себе не определяет, когда активность завершится: также должны учитываться зависимости от других активностей. Таким образом, время завершения каждой активности определяется оценкой объема работы для этой активности и временем, необходимым для перехода к этой активности в сетевом графике проекта. Время перехода к активности (или время, необходимое для подготовки к началу работы над этой активностью) равно максимальному времени по всем сетевым путям, ведущим

к этой активности. В более формальной записи время завершения активности *i* в проекте определяется по следующей рекурсивной формуле:

$$T_i = E_i + \text{Max}(T_{i-1},\ T_{i-2},\ ...,\ T_{i-n}),$$

где

- T_i — время завершения активности *i*;
- E_i — оценка объема работы для активности *i*;
- n — количество активностей, ведущих непосредственно к активности *i*.

Время для каждой из предшествующих активностей определяется тем же способом. Также можно воспользоваться регрессией, начать с последней активности в проекте и определить время завершения для каждой активности в сети. Для примера рассмотрим сеть активностей на рис. 7.6.

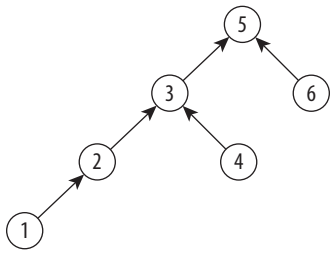


Рис. 7.6. Сетевой график проекта, использованный в примере вычисления времени

На диаграмме на рис. 7.6 активность 5 является последней. Таким образом, набор выражений регрессионных выражений, определяющих время завершения активности 5, выглядит так:

$$\begin{aligned}
T_5 &= E_5 + \text{Max}(T_3,\ T_6)\\
T_6 &= E_6\\
T_3 &= E_3 + \text{Max}(T_2,\ T_4)\\
T_4 &= E_4\\
T_2 &= E_2 + T_1\\
T_1 &= E_1
\end{aligned}$$

Обратите внимание: время завершения активности 5 зависит от оценки объема работ предыдущих активностей в такой же мере, в какой оно зависит от топологии сети. Например, если все активности на рис. 7.6 имеют одинаковую продолжительность:

$$T_5 = E_1 + E_2 + E_3 + E_5.$$

Но если для всех активностей, кроме шестой, оценка объема работы составляет 5 дней, а для шестой — 20 дней:

$$T_5 = E_6 + E_5 .$$

Хотя для небольших сетей, как на рис. 7.6, временные характеристики активностей могут быть вычислены вручную, в больших сетях вычисления быстро выходят из-под контроля. Компьютеры отлично справляются с регрессионными задачами, поэтому для вычисления времени завершения активностей можно воспользоваться соответствующими программами (например, Microsoft Project или электронной таблицей).

Критический путь

Вычисляя время завершения активностей, можно найти самый длинный возможный путь в сети активностей. В этом контексте «самым длинным путем» считается путь с наибольшей продолжительностью — не обязательно путь с наибольшим количеством активностей. Например, сетевой график проекта на рис. 7.7 состоит из 17 активностей, имеющих разную продолжительность (на рис. 7.7 указаны идентификаторы активностей, а не продолжительности).

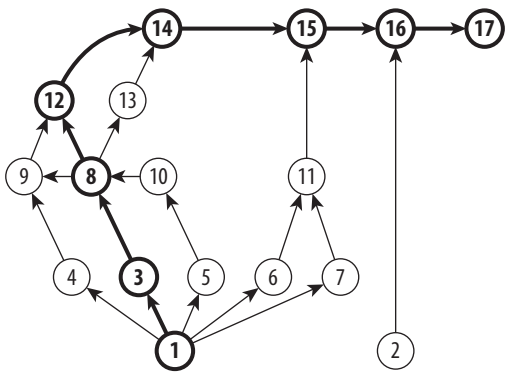


Рис. 7.7. Определение критического пути

На основании оценок объемов работы для всех активностей и их зависимостей, по приведенной выше формуле и начиная с активности 17, самый длинный путь в сети выделен жирными линиями. Самый длинный путь в сети называется **критическим путем**. Критический путь на сетевых графиках следует выделять линиями другого цвета или толщины. Определение критического

пути — единственный способ ответить на вопрос, сколько времени займет построение системы.

Так как критический путь является самым длинным путем в сети, он также определяет наименьшую возможную продолжительность проекта. Любая задержка на кратчайшем пути задерживает весь проект и ставит под угрозу ваши обязательства.

Ни один проект не может быть ускорен свыше критического пути. Иначе говоря, чтобы система строилась самым быстрым из возможных способов, она должна строиться по критическому пути. Это относится к любому проекту независимо от технологии, архитектуры, методологии разработки, процесса разработки, стиля управления и размера команды.

В любом проекте, состоящем из нескольких активностей, над которыми работает группа людей, существует сеть активностей с критическим путем. Для критического пути неважно, знаете вы о его существовании или нет; он просто существует. Без анализа критического пути вероятность того, что разработчики построят систему по критическому пути, близка к нулю. Скорее всего, с таким подходом работа займет гораздо больше времени.

> **ПРИМЕЧАНИЕ** Хотя в обсуждении до настоящего момента критический путь проекта упоминался в единственном числе, проект вполне может иметь несколько критических путей (имеющих равную продолжительность); более того, все сетевые пути могут оказаться критическими. Проекты с несколькими критическими путями сопряжены с высоким риском, потому что любая задержка на любом из этих путей приведет к задержке проекта.

Назначение ресурсов

В ходе планирования проекта архитектор назначает абстрактные ресурсы (например, *Разработчик 1*) для каждого из вариантов плана проекта. Только после того, как ответственные за принятие решений выберут конкретный вариант проекта, менеджер проекта сможет назначить фактические ресурсы. Так как любая задержка на критическом пути приведет к задержке всего проекта, менеджер проекта всегда должен сначала назначать ресурсы на критическом пути. Вам стоит пойти еще дальше и всегда сначала назначать на критический путь свои *лучшие* ресурсы. Под «лучшими» я имею в виду самых надежных и заслуживающих доверия разработчиков — тех, которые не подведут. Не совершайте классическую ошибку, при которой разработчики сначала назначаются на заметные, но некритичные активности, или на активности, которые наиболее важны для заказчиков или руководства. Первоочередное назначение ресурсов разработки на некритические активности никак не способствует ускорению проекта. Замедление на критическом пути безусловно приводит к замедлению всего проекта.

Численность персонала

В ходе планирования проекта для каждого варианта плана проекта архитектор должен определить, сколько ресурсов (таких, как разработчики) потребуется для проекта в целом. Архитектор определяет необходимую численность персонала итеративным методом. Возьмем сеть на рис. 7.7, где критический путь уже определен; предположим, каждый узел соответствует сервису. Сколько разработчиков потребуется в первый день проекта? Если в вашем распоряжении всего один разработчик, то этот разработчик по определению является вашим лучшим разработчиком, поэтому он назначается на активность 1. Если в вашем распоряжении два разработчика, вы можете назначить второго на активность 2, хотя эта активность потребуется намного позднее. Если в вашем распоряжении три разработчика, то третий разработчик в лучшем случае простаивает, а в худшем мешает разработчику, работающему над активностью 1. Таким образом, правильный ответ на вопрос о том, сколько разработчиков потребуется в день 1 проекта, — не более двух разработчиков.

Теперь предположим, что активность 1 завершена. Сколько разработчиков потребуется теперь? Ответ: не более шести (доступны активности 3, 4, 5, 6, 7 и 2). Однако запрашивать шесть разработчиков не стоит, потому что к тому моменту, когда вы пройдете по критическому пути до уровня активностей 8 и 12, вам понадобятся только три или даже два разработчика. Возможно, после завершения активности 1 лучше запросить всего четырех разработчиков вместо шести. Использование только четырех разработчиков вместо шести имеет два значительных преимущества. Во-первых, вы сокращаете стоимость проекта. Проект с четырьмя разработчиками обойдется на 33% дешевле, чем проект с шестью разработчиками. Во-вторых, команда из четырех разработчиков намного эффективнее команды из шести разработчиков. В меньшей команде будет меньше коммуникационного шума и меньше вмешательства со стороны тех, кому нечем заняться.

Если руководствоваться только этим критерием, команда из трех и даже двух разработчиков будет лучше команды из четырех разработчиков. Тем не менее анализ сети на рис. 7.7 показывает, что систему, скорее всего, невозможно построить только с тремя разработчиками при сохранении той же продолжительности работы. При столь малом количестве разработчиков вы сами загоняете себя в угол: разработчику на критическом пути может понадобиться некритическая активность, которая еще попросту не готова (например, активности 15 необходима активность 11). Таким образом, некритическая активность становится критической, что, по сути, приводит к созданию нового критического пути большей длины. Я называю такую ситуацию *субкритическим комплектованием*. Когда проект переходит в субкритическое состояние, срок будет нарушен, потому что старый критический путь уже недействителен.

Настоящий вопрос заключается не в том, сколько потребуется ресурсов. Вопрос, который следует себе задавать в любой фазе проекта, звучит так:

Какой наименьший уровень ресурсов позволит проекту беспрепятственно продвигаться по критическому пути?

Поиск наименьшего уровня ресурсов обеспечивает критическое комплектование проекта в любой момент времени и позволяет реализовать проект наименее затратным и самым эффективным образом. Обратите внимание: критический уровень комплектования может и должен изменяться на протяжении жизненного цикла проекта.

Представьте группу разработчиков без плана проекта. Вероятность того, что эта группа составляет наименьший уровень ресурсов, необходимый для беспрепятственного продвижения проекта по критическому пути, близка к нулю. Единственный способ компенсировать неизвестные потребности проекта в комплектовании — применение невероятно расточительного и неэффективного избыточного комплектования. Как было показано ранее, такая организация работы не может быть самым быстрым способом завершения проекта, а теперь вы видите, что она также не может быть самым низкозатратным способом построения системы. Мой опыт показывает, что избыточное комплектование может превышать минимально необходимый уровень в несколько раз.

Временной резерв

Вернемся к сети на рис. 7.7. После того как вы заключили, что систему можно попытаться построить всего с четырьмя разработчиками, появляется новая проблема: где и когда следует задействовать этих четырех разработчиков? Например, при завершенной активности 1 можно назначить разработчиков на активности 3, 4, 5, 6, или 3, 5, 6, 7, или 3, 4, 6, 2, и т. д. Даже с простой сетью комбинаторный спектр возможностей слишком велик. А ведь каждый из этих вариантов имеет собственный набор комбинаций при последующих назначениях.

К счастью, перебирать эти комбинации не придется. Присмотритесь к активности 2 на рис. 7.7. Назначение ресурсов активности 2 можно отложить до начала дня, в который должна начаться активность 16 (находящаяся на критическом пути), за вычетом оцениваемой продолжительности активности 2. Активность 2 может «плавать» (оставаться без назначений и не запускаться) до того момента, когда она соприкоснется с активностью 16. Все некритические активности имеют *временной резерв*, который определяется как промежуток времени, на который можно отложить их выполнение без задержки проекта. У критических активностей временного резерва нет (вернее, их временной резерв равен 0), поскольку любая задержка этих активностей приведет к задержке всего проекта. При назначении ресурсов в проекте руководствуйтесь следующим правилом:

Всегда назначайте ресурсы на основании временного резерва.

КЛАССИЧЕСКАЯ ЛОВУШКА

Как заметил Том Демарко[1], многие организации подталкивают своих руководителей к выполнению неверных действий при комплектовании проектов, даже если это делается с лучшими намерениями. Руководители могут правильно назначить разработчиков на проект только после планирования проекта, которое возможно только после завершения архитектуры. Эти операции занимают не много времени по своей природе, но они завершают нечеткую начальную стадию проекта, которая сама по себе может занять месяцы: определение объема работы, прототипизация, оценка технологий, собеседования с заказчиками, анализ требований и т. д. Нет смысла нанимать разработчиков до того момента, когда менеджер проекта сможет назначать их на основании плана, потому что в противном случае им будет нечего делать. Однако офисы и столы, иногда пустующие месяцами, плохо влияют на руководителя: может показаться, что он просто бездельничает. Руководитель боится, что когда (а не если) проект задержится (как это обычно бывает с программными проектами), его обвинят в этом, потому что он не нанял разработчиков в начале проекта. Чтобы избежать этих претензий, руководитель нанимает разработчиков в самом начале нечеткой начальной стадии, чтобы офисы не пустовали. Разработчикам нечем заняться, поэтому они проводят время за играми, чтением блогов и долгими обеденными перерывами. К сожалению, такое поведение отражается на руководителе еще хуже, чем пустые офисы, потому что теперь складывается впечатление, что руководитель не умеет управлять и поручать работу, а расплачиваться за это приходится организации.

Руководитель снова боится, что если проект задержится, то он снова окажется виновным. Как только начинается начальная фаза, руководитель комплектует проект и поручает функцию A первому разработчику, функцию B — второму и т. д., хотя у проекта еще нет ни основательной архитектуры, ни анализа критического пути. Когда через несколько недель или месяцев архитектор выдает архитектуру и план проекта, они оказываются неактуальными, потому что разработчики трудились над совершенно другой системой и проектом. Проект серьезно нарушает график и превышает установленный бюджет не только из-за отсутствия архитектуры и анализа критического пути, но также потому, что происходившее в начале было функциональной декомпозицией системы и функциональной декомпозицией команды.

То, что говорилось в главе 2 о декомпозиции системы, также легко адаптируется к декомпозиции команд. В проекте теперь появляется худшая из возможных комбинаций проектирования системы и планирования команды. Руководитель постоянно выпрашивает у высшего начальства дополнительное время

[1] Tom Demarco, The Deadline (Dorset House, 1997) (на русском: *Том Де Марко*. Deadline. Роман об управлении проектами. М.: Вершина, 2006. — *Примеч. ред.*).

и ресурсы. Когда проект запаздывает (как большинство проектов в области разработки), руководитель выглядит не хуже, чем любой другой руководитель в организации.

Действовать правильно намного проще во второй раз, когда вы уже доказали свою способность завершать проекты в заданный срок и в рамках бюджета. Возможно, организация не поймет, как это произошло (или почему не работает то, что пытаются делать другие руководители), но не сможет спорить с результатами. Но когда вы впервые идете по этому пути, не имея истории успеха, вам будет нелегко. Лучшее, что вы можете сделать, — противостоять проблеме и сделать ее решение частью вашего планирования проекта, как было описано в главе 11.

Чтобы определить, как назначить разработчиков в приведенном примере после завершения активности 1, вычислите временной резерв всех активностей, которые становятся возможными после завершения активности 1, и назначьте четырех разработчиков по возрастанию временного резерва. Сначала назначьте разработчика на критический путь — не из-за его особой роли, а потому, что он имеет наименьший возможный временной резерв. Теперь допустим, что активность 2 имеет 60 дней временного резерва, а активность 4 — только 5 дней. Это означает, что если начало активности 4 будет отложено более чем на 5 дней, это приведет к нарушению графика проекта. С другой стороны, назначение на активность 2 можно отложить до 60 дней, поэтому следующий разработчик назначается на активность 4. В промежуточное время, в которое активность 2 остается неукомплектованной, вы фактически потребляете временной резерв этой активности. Возможно, к тому времени, когда резерв активности 2 сократится до 15 дней, вы наконец-то сможете назначить разработчика на эту активность.

Этот процесс имеет итеративную природу, потому что изначально наименьший уровень комплектования неизвестен, а назначение с применением временного резерва изменяет временные резервы активностей. Для начала попытайтесь укомплектовать проект некоторым уровнем ресурсов (например, шестью ресурсами), а затем назначьте эти ресурсы на основании временных резервов. Каждый раз, когда вы планируете назначение ресурса для завершения активностей, найдите в сети ближайшие доступные активности и выберите активность с наименьшим временным резервом в качестве следующего назначения для этого ресурса. Если вам удалось укомплектовать проект, попробуйте еще раз — на этот раз с сокращенным уровнем ресурсов (например, пятью или четырьмя ресурсами). В какой-то момент у вас появится избыток активностей по сравнению с доступными ресурсами. Если эти неукомплектованные активности имеют достаточно высокий временной резерв, отложите назначение ресурсов до того момента, когда ресурсы будут доступны. Пока эти активности не укомплектованы, вы будете потреблять их временной ре-

зерв. Если активности станут критическими, вы не сможете построить проект с таким уровнем комплектования и вам придется остановиться на более высоком уровне ресурсов.

Другое ключевое преимущество комплектования на основе временного резерва связано с сокращением рисков. Активности с наименьшим временным резервом — самые рискованные, в наибольшей степени способные вызвать задержку проекта. Назначение ресурсов этим активностям сначала позволит вам укомплектовать проект ресурсами наиболее безопасным образом, а также сократит общий риск, связанный с любым заданным уровнем комплектования. И снова без планирования проекта вероятность того, что менеджер проекта или группы разработчиков распределит ресурсы между активностями на основании временного резерва, близка к нулю. Такой подход получается не только медленным и дорогостоящим, но и рискованным.

Сеть и ресурсы

Обсуждение до настоящего момента было сосредоточено на зависимостях между активностями как механизма построения сети. Тем не менее ресурсы также влияют на сеть. Например, если поручить сеть, изображенную на рис. 7.7, одному разработчику, фактическая диаграмма сети будет напоминать длинную цепочку, а не рис. 7.7. Зависимость от одного ресурса кардинально изменяет сетевую диаграмму. Таким образом, сетевая диаграмма представляет собой не сеть активностей, а в первую очередь сеть зависимостей. Если вы располагаете неограниченными ресурсами и используете исключительно гибкую схему комплектования, тогда вы можете полагаться только на зависимости между активностями. После того как вы начнете потреблять временной резерв, зависимости от ресурсов необходимо добавить в сеть. Ключевое наблюдение выглядит так:

Зависимости от ресурсов также являются зависимостями.

Фактический способ назначения ресурсов в сетевом графике проекта определяется несколькими переменными. При назначении ресурсов необходимо принять во внимание следующие факторы:

- Предпосылки планирования.
- Критический путь.
- Временные резервы.
- Доступные ресурсы.
- Ограничения.

Эти факторы всегда приводят к появлению нескольких вариантов планирования, даже в относительно прямолинейных проектах.

Планирование активностей

Комбинация сетевого графика проекта, критического пути и анализа временных резервов позволяет вычислить продолжительность проекта, а также моменты начала каждой активности относительно начала проекта. Тем не менее информация в сети исчисляется в рабочих днях, а не в календарных датах. Чтобы преобразовать информацию из сетевого графика в календарные даты, необходимо произвести планирование активностей. Эта задача легко решается при помощи таких программ, как Microsoft Project. Определите все активности в программе, добавьте зависимости в виде предшественников и назначьте ресурсы в соответствии со своим планом. После того как вы выберете начальную дату для проекта, программа составит график всех активностей. Выходные данные также могут включать диаграмму Ганта, но это не имеет прямого отношения к базовой информации, которую можно получить при помощи программы: запланированные даты начала и завершения для каждой активности в проекте.

> **ВНИМАНИЕ** Диаграммы Ганта, изолированные от других данных, вредны для проекта, потому что могут создать иллюзию планирования и контроля. Диаграмма Ганта — всего лишь одно из представлений сетевого графика проекта, и она не включает полный план проекта.

Распределение кадров

Потребности проекта в кадрах изменяются с течением времени. Сначала потребуется только основная команда. После того как управление выберет вариант плана проекта и утвердит проект, вы сможете добавить другие ресурсы (например, разработчиков и тестировщиков).

Не все ресурсы потребуются одновременно из-за существования зависимостей и критического пути. Аналогичным образом не все ресурсы освобождаются в постоянном темпе. Основная команда необходима на всем протяжении проекта, но разработчики не обязательно должны быть задействованы до последнего дня проекта. В идеале они должны постепенно появляться от начала проекта, когда все больше активностей становятся доступными, и выводиться из проекта ближе к его завершению.

Такой подход постепенного введения и выведения ресурсов имеет два значимых преимущества. Во-первых, он избегает циклов «избыток/нехватка», с которыми сталкиваются многие программные проекты. Даже если вы располагаете требуемым средним уровнем кадров для проекта, в одних ресурсах может наблюдаться нехватка, а в других — избыток. Такие циклы простоев и авралов чрезвычайно неэффективны и вызывают моральное разложение в коллективе. Во-вторых (что важнее), постепенное введение ресурсов предоставляет воз-

можность экономии на масштабах. Если организация ведет несколько проектов, вы можете организовать их так, чтобы разработчики всегда выводились из одного проекта при введении в другой проект. Такой режим работы способен обеспечить многократный прирост производительности, классическое проявление принципа «добиваться большего меньшими средствами».

> **ПРИМЕЧАНИЕ** Постепенное введение и выведение ресурсов между проектами базируется на основательном проектировании системы и планировании проекта, которые гарантируют целостную структуру системы, отделяющую конкретных разработчиков от конкретных компонентов.

Диаграмма распределения кадров

На рис. 7.8 изображена типичная диаграмма распределения кадров для хорошо спроектированного и правильно укомплектованного проекта. Проект открывается начальной стадией, во время которой основная команда работает над проектированием системы и планированием проекта; эта стадия завершается анализом SDP (Software Development Plan, «план разработки программного продукта»). Если проект будет завершен в этой точке, комплектование падает до нуля, а основная команда становится доступной для других проектов. Если проект будет утвержден, происходит исходный рост кадров, когда разработчики и другие ресурсы работают над активностями самого нижнего уровня, обеспечивающими возможность реализации других активностей. Когда эти активности становятся доступными, проект может принять дополнительное комплектование. В какой-то момент будут введены все ресурсы, которые могут понадобиться проекту; достигается пиковое комплектование. На какое-то время проект полностью укомплектован. Обычно система формируется к концу этой

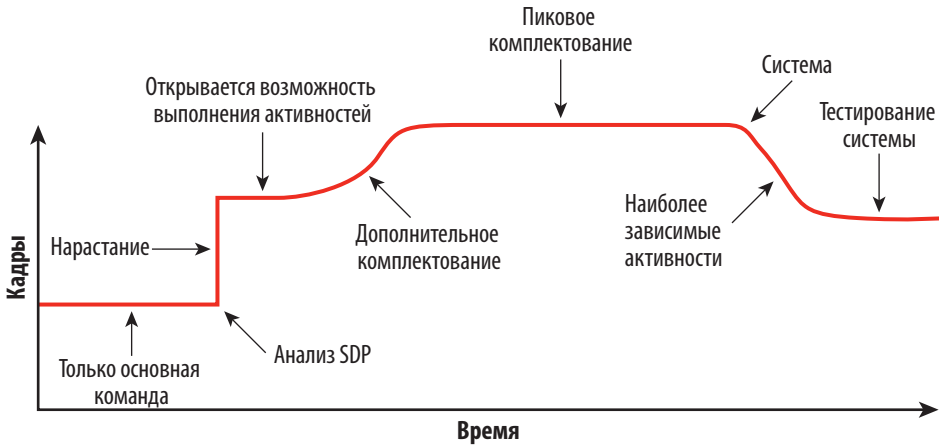


Рис. 7.8. Правильное распределение кадров

фазы. Теперь проект может постепенно выводить ресурсы, а остающиеся ресурсы используются для работы над самыми зависимыми активностями. Проект завершается с уровнем кадров, необходимым для тестирования и выпуска системы.

На рис. 7.9 изображена диаграмма распределения комплектования, демонстрирующая поведение рис. 7.8. Чтобы построить такую диаграмму, как на рис. 7.9, сначала укомплектуйте проект, а затем перечислите все даты, представляющие интерес (уникальные даты начала и завершения активностей), в хронологическом порядке. Вы можете подсчитать, сколько ресурсов требуется для каждой категории ресурсов, в любой период времени между интересующими датами. Не забывайте включать в распределение ресурсы, которые не связаны с конкретными активностями, но при этом являются обязательными: основная команда, контроль качества и разработчики между активностями, связанными с программированием. Подобные столбцовые диаграммы элементарно строятся в электронных таблицах. В файлах, прилагаемых книге, содержатся примеры проектов и шаблоны для таких диаграмм.

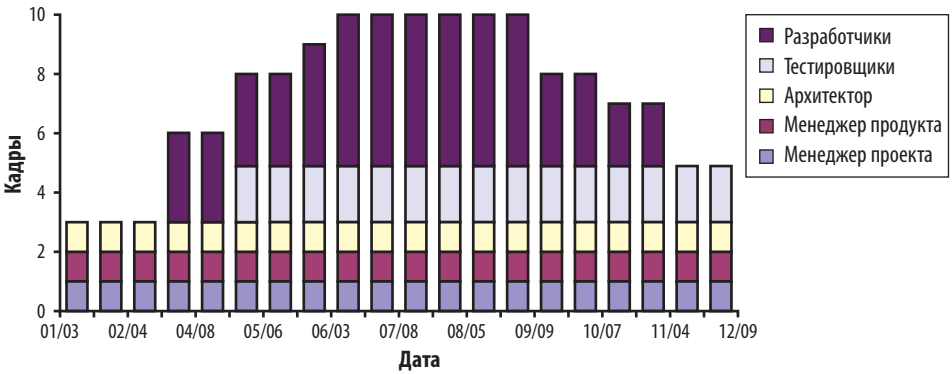


Рис. 7.9. Пример нормального распределения кадров

Так как даты, представляющие интерес, могут быть разделены разными интервалами, у диаграмм распределения комплектования плотность по оси времени может изменяться. Тем не менее в большинстве проектов сколько-нибудь серьезного размера с достаточным количеством активностей общая форма диаграммы должна примерно соответствовать форме на рис. 7.8. Анализируя диаграмму распределения кадров, можно быстро получить ценную информацию о качестве планирования проекта.

Ошибки комплектования

На диаграмме распределения кадров могут очевидно проявляться некоторые стандартные ошибки комплектования. Если диаграмма имеет прямоугольную

форму, это указывает на постоянное комплектование — я уже предупреждал о нежелательности этой практики.

Распределение кадров с громадным пиком посередине (как на рис. 7.10) также является плохим признаком: такой пик всегда свидетельствует о неэффективности.

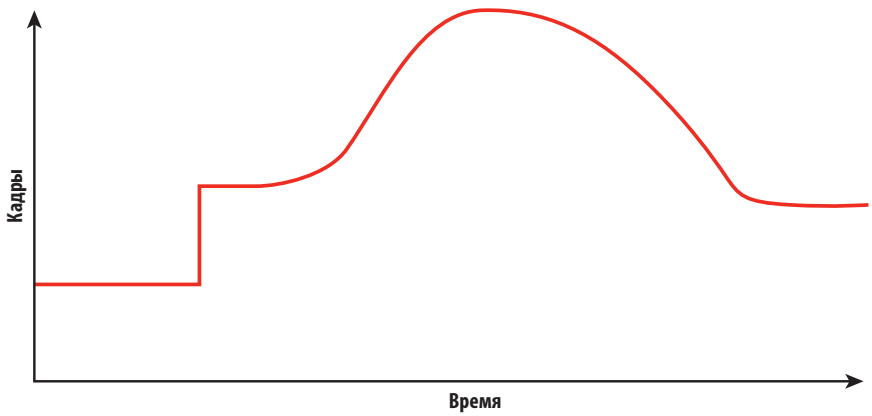


Рис. 7.10. Пик в распределении кадров

Примите во внимание усилия, затраченные на наем людей и их обучение в предметной области, архитектуре и технологии, когда вы собираетесь использовать их в течение короткого периода времени. Присутствие пика обычно обусловлено недостаточным потреблением временного резерва в проекте, что приводит к пиковой потребности в ресурсах. Если бы проект расходовал часть временного резерва вместо ресурсов, кривая была бы более плавной. На рис. 7.11 изображен тот же проект с пиком комплектования.

Горизонтальная линия на диаграмме распределения кадров (как на рис. 7.12) — еще одна классическая ошибка. Она указывает на отсутствие «плоскогорья» на рис. 7.8. Скорее всего, проект находится в субкритическом состоянии и ему не хватает ресурсов для комплектования некритических активностей исходного плана.

На рис. 7.13 изображено распределение кадров для субкритического проекта. Этот проект переходит в субкритическое состояние на уровне 11 или 12 ресурсов. Мало того что на нем нет «плоскогорья» — на его месте находится «долина».

Хаотичные колебания в распределении кадров (как на рис. 7.14) — еще один тревожный сигнал. Проекты, спроектированные настолько «эластично», обречены на неудачу (рис. 7.15), потому что комплектование не может быть *настолько* эластичным. В большинстве проектов невозможно сотворить людей

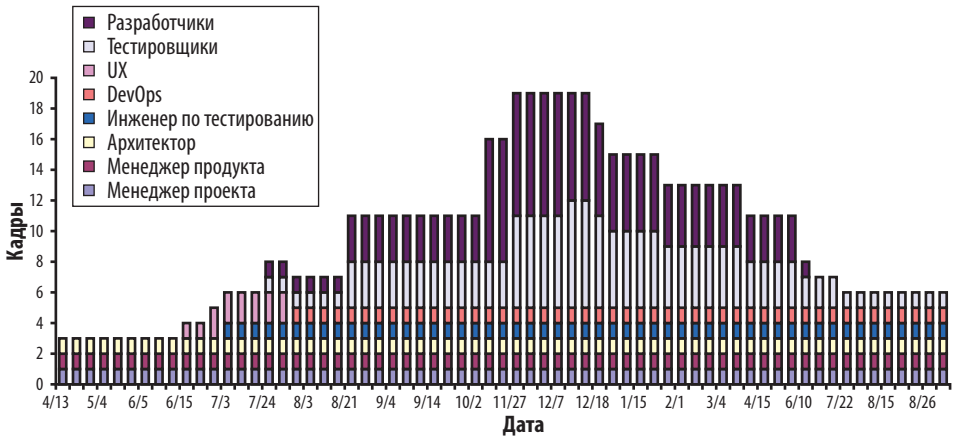


Рис. 7.11. Пример пика в распределении кадров

Рис. 7.12. Горизонтальное субкритическое распределение кадров

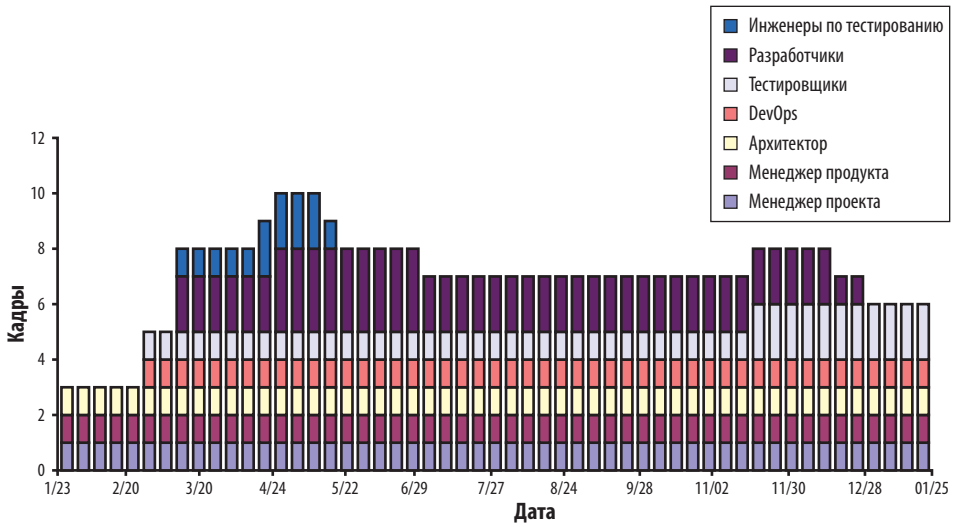


Рис. 7.13. Пример субкритического распределения кадров

из воздуха, моментально добиться от них результативности, а потом избавиться от них через минуту. Кроме того, когда люди постоянно приходят и уходят из проекта, их обучение (или переобучение) обходится очень дорого. Трудно возлагать ответственность на людей или сохранять их квалификацию в таких обстоятельствах.

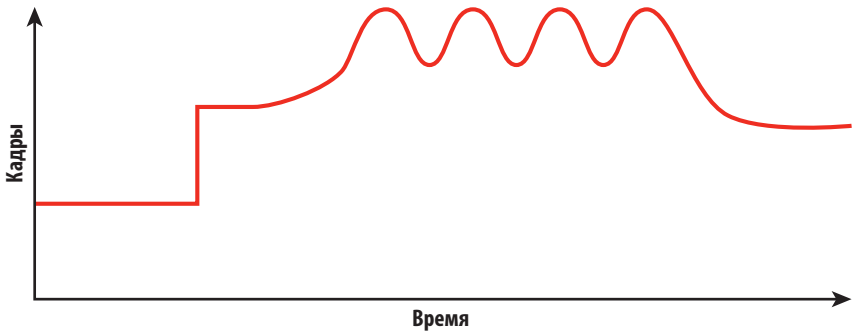


Рис. 7.14. Хаотическое распределение кадров

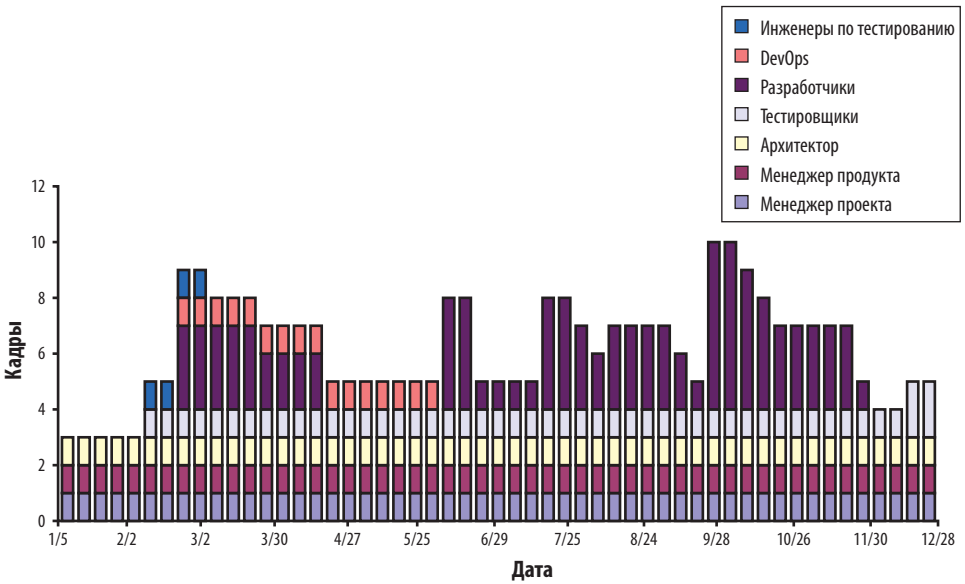


Рис. 7.15. Пример хаотического распределения кадров

На рис. 7.16 показано другое распределение кадров, которого следует избегать, — резкое наращивание в начале проекта. Хотя на диаграмме не приведены какие-либо числа, на ней очевидно проявляется ложный оптимизм. Никакая

команда не сможет моментально перейти от нуля к пиковому комплектованию, чтобы все создавали высококачественный код коммерческого уровня. Даже если в проекте изначально ведется такой объем параллельной работы и даже если вы располагаете необходимыми ресурсами, последующая часть сети отрегулирует количество ресурсов, которые могут быть поглощены проектом, и требуемая комплектация постепенно иссякает.

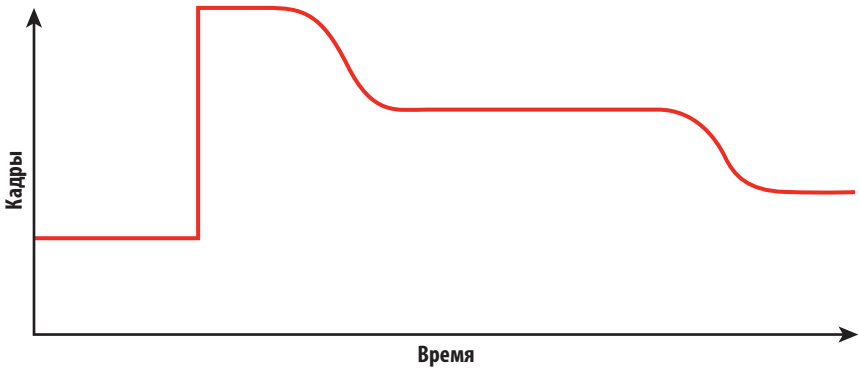


Рис. 7.16. Резкое наращивание кадров

На рис. 7.17 продемонстрирован такой проект. План рассчитан на мгновенное получение 11 человек, а потом вскоре сдувается до шести человек до конца проекта. Маловероятно, что любая команда сможет разогнаться таким образом, а доступные ресурсы используются неэффективно из-за слишком большого размера команды.

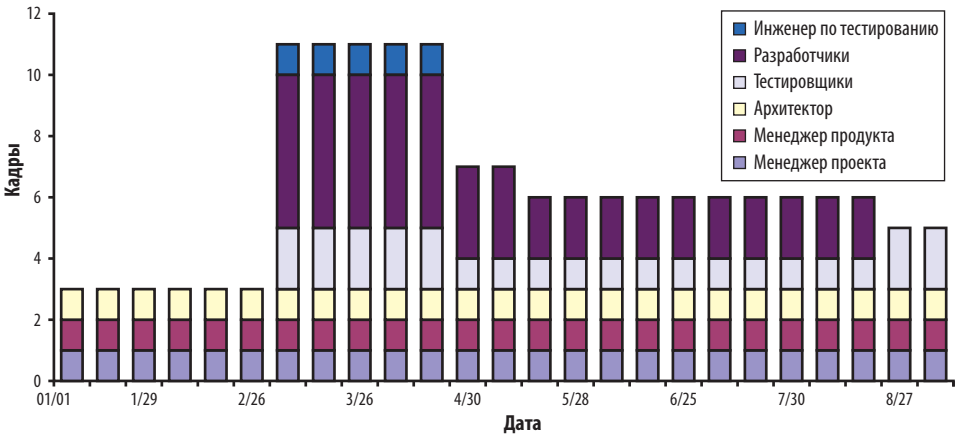


Рис. 7.17. Пример резкого наращивания кадров

Сглаживание кривой

Главный вывод, который можно сделать из визуальных признаков ошибок на диаграммах: что хорошие проекты имеют плавное распределение кадров. Жизнь намного приятнее, когда проект движется в стабильном темпе, без резких поворотов, неожиданных разгонов и экстренных торможений.

Как упоминалось ранее, две корневые причины ошибочного комплектования — ожидание слишком эластичного комплектования и недостаточное потребление временного резерва при назначении ресурсов. При рассмотрении эластичности кадров вы должны хорошо знать вашу команду и хорошо понимать, на что она способна с точки зрения доступности и эффективности. Степень эластичности кадров также зависит от природы организации, качества системы и планирования проекта. Чем выше качество проектирования, тем быстрее разработчики осваиваются с новой системой и активностями. Потребление временного резерва легко реализуется в большинстве проектов; оно с большой вероятностью сократит как нестабильность в комплектовании, так и абсолютный уровень требуемого комплектования. Более реалистичный подход к эластичности комплектования и потреблению временного резерва часто устраняет пики, подъемы/падения и быстрое наращивание.

> **ПРИМЕЧАНИЕ** Не путайте диаграмму распределения кадров (без расширения проекта или его затрат) с выравниванием нагрузки — расширением продолжительности проекта в соответствии с более низким уровнем ресурсов. Выравнивание нагрузки — другое название для субкритического комплектования (так, как оно определяется в этой главе).

Затраты на реализацию проекта

Построение диаграммы распределения комплектования для каждого варианта планирования проекта — замечательный инструмент для проверки варианта на здравый смысл. Если при планировании проекта что-то кажется неправильным, то, как правило, так оно и есть.

Диаграмма распределения комплектования предоставляет другое очевидное преимущество: она помогает вычислить затраты на реализацию проекта. В отличие от физических строительных проектов, в программные проекты не включается стоимость строительных материалов или товаров. Затраты на разработку программного проекта в подавляющем большинстве определяются затратами на персонал. К этой категории относятся все участники команды, от основной команды до разработчиков и тестировщиков. Затраты на персонал просто равны уровню комплектования, умноженному на время:

$$\text{Затраты} = \text{уровень комплектования} \times \text{время}.$$

Умножение комплектования на время в действительности дает площадь под кривой распределения комплектования. Чтобы вычислить затраты, необходимо вычислить эту площадь.

Диаграмма распределения комплектования является дискретной моделью проекта с вертикальными столбцами (уровень комплектования) за каждый период времени между датами, представляющими интерес. Площадь под диаграммой распределения комплектования вычисляется умножением высоты каждого вертикального столбца (количество людей) на продолжительность периода времени между датами (рис. 7.18). Затем полученные произведения суммируются.

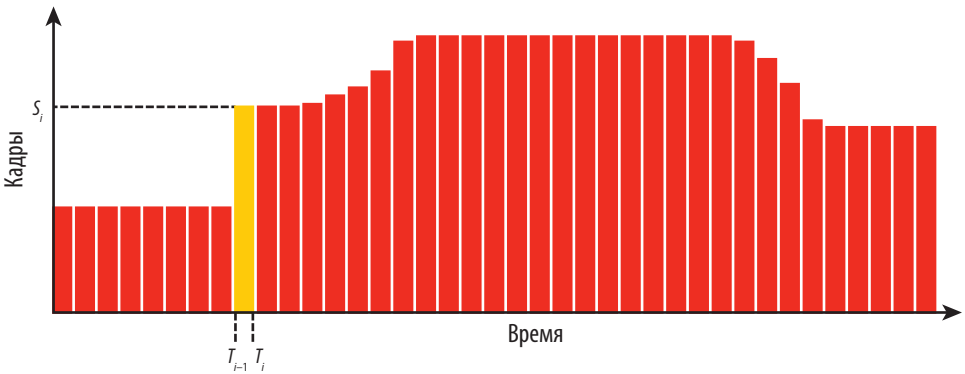


Рис. 7.18. Вычисление затрат на реализацию проекта

Площадь под диаграммой комплектования вычисляется по следующей формуле:

$$\text{Затраты} = \sum_{i=1}^{n} (S_i \times (T_i - T_{i-1})),$$

где:

- S_i — уровень комплектования для i-й даты;
- T_i — i-я дата, представляющая интерес (T_0 — дата начала);
- n — количество дат в проекте.

Вычисление площади под диаграммой распределения комплектования — единственный способ получить ответ на вопрос о том, в какую сумму обойдется проект.

Если вы используете электронную таблицу для построения диаграммы распределения комплектования, для вычисления площади под диаграммой достаточно добавить еще один столбец с накапливаемой суммой (по сути, речь

идет о числовом интегрировании). В файлах, прилагаемых к книге, содержатся примеры таких вычислений.

Так как затраты определяются как уровень комплектования, умноженный на время, они должны измеряться в единицах объема работ и времени (например, в человеко-месяцах или человеко-годах). Лучше использовать эти единицы вместо денежных величин, чтобы нейтрализовать различия в уровнях зарплат, локальных валютах и бюджетах. Это позволит объективно сравнивать затраты на разные варианты планирования проекта.

Для заданной архитектуры, исходного распределения заданий и оценки трудозатрат для получения ответов на вопросы о том, сколько времени займет работа и в какую сумму обойдется построение системы, требуется несколько часов, максимум день. К сожалению, большинство программных проектов движется вслепую. Это так же разумно, как играть в покер, не заглянув в карты, — только вместо фишек на кону ваш проект, ваши карьерные перспективы и даже будущее компании.

Когда затраты на реализацию проекта известны, можно вычислить эффективность проекта. Эффективность проекта определяется как отношение суммы объемов работ по всем активностям (предполагается, что персонал используется идеально) к фактическим затратам на реализацию проекта. Например, если сумма объемов работ по всем активностям составляет 10 человеко-месяцев (предполагается, что месяц состоит из 30 рабочих дней), а затраты на проект равны 50 человеко-месяцам, эффективность проекта составляет 20%.

Эффективность проекта — превосходный показатель качества планирования проекта. Ожидаемая эффективность хорошо спроектированной системы при хорошо спланированном и укомплектованном проекте лежит в диапазоне от 15 до 25%.

Такой диапазон эффективности может показаться ужасно низким, но более высокая эффективность обычно является верным признаком нереалистичного планирования проекта. Ни один процесс в природе не может даже приблизиться к 100% эффективности. Ни один проект не свободен от ограничений, и эти ограничения не позволяют вам использовать ресурсы самым эффективным образом. А если добавить все затраты на основную команду, тестировщиков, на построение и DevOps, а также все остальные ресурсы, связанные с вашим проектом, доля трудозатрат, связанных непосредственно с написанием кода, сильно сокращается. Проекты с эффективностью около 40% построить просто невозможно.

Даже 25-процентная эффективность немного завышена; она достигается с правильной архитектурой системы, обеспечивающей проект самой эффективной командой (см. рис. 7.1) и правильным планом проекта, который использует минимальный уровень ресурсов и назначает их с учетом временных резервов.

Дополнительные факторы, необходимые для завершения проекта при высокой ожидаемой эффективности, — небольшая опытная команда, участники которой привыкли работать вместе, и менеджер проекта, серьезно относящийся к качеству и способный справиться со сложностью проекта.

Эффективность также связана с эластичностью комплектования. Если комплектование было действительно эластичным (то есть вы всегда можете получить ресурсы именно тогда, когда они нужны), эффективность будет высокой. Конечно, комплектование никогда не бывает *настолько* эластичным, поэтому иногда даже назначенные на проект ресурсы будут простаивать, что приведет к снижению эффективности. Это особенно справедливо при использовании ресурсов за пределами критического пути. Если один человек работает над всеми критическими активностями, обычно этот человек работает на пике эффективности, потому что сразу после завершения одной активности он берется за другую, и затраты на эту работу приближаются к сумме затрат критических активностей. С некритическими активностями всегда существует некоторый временной резерв. Поскольку комплектование никогда не бывает настолько эластичным, ресурсы за пределами критического пути никогда не могут использоваться с очень высокой эффективностью.

Если вариант планирования проекта обладает очень высокой эффективностью, вы должны проанализировать истинную причину. Возможно, ваши предположения относительно полноты и эластичности комплектования были слишком оптимистичными или же сетевой график проекта оказался слишком критическим. В конце концов, если большинство сетевых путей либо критичны, либо почти критичны (большинство активностей имеет низкий временной резерв), то вы получите высокий показатель эффективности. Однако у такого проекта очевидно существует высокий риск нарушения исходных обязательств.

Эффективность как общая оценка

Эффективность программных проектов тесно связана с природой организации. Неэффективные организации не становятся эффективными за один вечер, и наоборот. Эффективность также связана с природой бизнеса. Избыточные затраты, необходимые для проекта, разрабатывающего программы для медицинских устройств, отличаются от затрат небольшой начинающей фирмы, разрабатывающей плагины для социальных сетей.

Эффективность может использоваться как еще одна общая оценка проекта. Допустим, вы знаете, что исторически ваши проекты обладали 20-процентной эффективностью. После того как вы получите исходный список активностей и их оценки, просто умножьте сумму объемов работ (в предположении об идеальной эффективности использования) по всем активностям на 5, чтобы получить приблизительную оценку затрат на реализацию всего проекта.

Планирование освоенной ценности

Другой информативный метод планирования проектов — *планирование освоенной ценности*. Освоенная ценность является популярным средством отслеживания состояния проекта, но она также может использоваться как отличное средство планирования проектов. При планировании освоенного объема каждой активности назначается определенная ценность в отношении завершения проекта, а затем объединение графиков всех активностей показывает, как вы планируете реализовывать эту ценность как функцию времени.

Формула для запланированной освоенной ценности выглядит так:

$$EV(t) = \frac{\sum_{i=1}^{m} E_i}{\sum_{i=1}^{N} E_i},$$

где:

- E_i — оцениваемая продолжительность для i-й активности;
- m — количество активностей, завершенных ко времени t;
- N — количество активностей в проекте;
- t — точка на шкале времени.

Освоенная ценность на момент t равна отношению суммы оцениваемой продолжительности всех активностей, завершенных ко времени t, к сумме оцениваемых продолжительностей всех активностей.

Для примера возьмем очень простой проект из табл. 7.1.

Таблица 7.1. Освоенная ценность проекта

Активность	Продолжительность (в днях)	Ценность (%)
Начальная стадия	40	20
Сервис доступа	30	15
UI	40	20
Сервис менеджера	20	10
Вспомогательные средства	40	20
Тестирование системы	30	15
Итого	200	100

Сумма оцениваемой продолжительности для всех активностей в табл. 7.1 составляет 200 дней. Например, оценка продолжительности для активности UI составляет 40 дней. Так как 40 составляет 20% от 200, можно утверждать, что

при завершении активности UI дает 20% к завершению проекта. Из графика реализации активностей вы также знаете, когда должна быть завершена активность UI, так что вы можете вычислить планируемое освоение ценности как функцию времени (табл. 7.2).

Таблица 7.2. Пример планирования осваиваемой ценности как функции времени

Активность	Дата завершения	Ценность (%)	Освоенная ценность (%)
Запуск	0	0	0
Начальная стадия	t_1	20	20
Сервис доступа	t_2	15	35
UI	t_3	20	55
Сервис менеджера	t_4	10	65
Вспомогательные средства	t_5	20	85
Тестирование системы	t_6	15	100

Диаграмма планируемого прогресса изображена на рис. 7.19. К тому моменту, когда проект достигнет запланированной даты завершения, он должен освоить 100% ценности. Главный вывод из рис. 7.19 заключается в том, что наклон кривой планируемой осваиваемой ценности представляет результативность работы команды. Если поручить тот же проект более квалифицированной команде, они скорее реализуют 100% освоенной ценности и у них линия на графике будет более крутой.

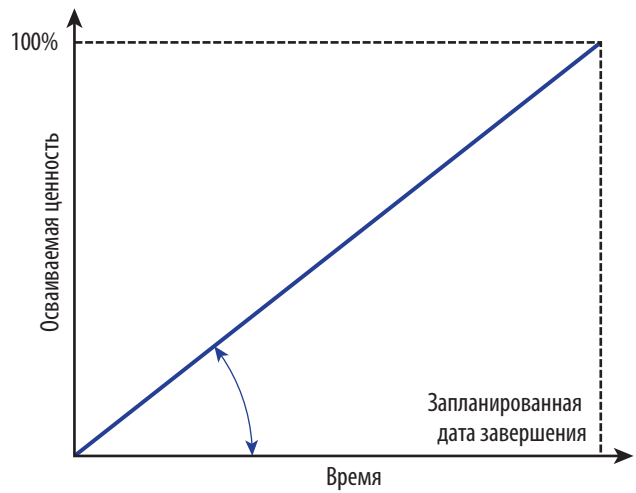


Рис. 7.19. Диаграмма планируемой осваиваемой ценности

Классические ошибки

Осознание того, что ожидаемую результативность работы команды можно оценить по диаграмме осваиваемой ценности, позволяет быстро обнаружить ошибки в плане проекта. Для примера возьмем диаграмму планируемой осваиваемой ценности на рис. 7.20. Ни одна команда в мире не сможет завершить работу по такому плану. Большую часть проекта ожидаемая результативность была невысокой. Какое чудо продуктивности обеспечит стремительный взлет осваиваемой ценности к концу проекта?

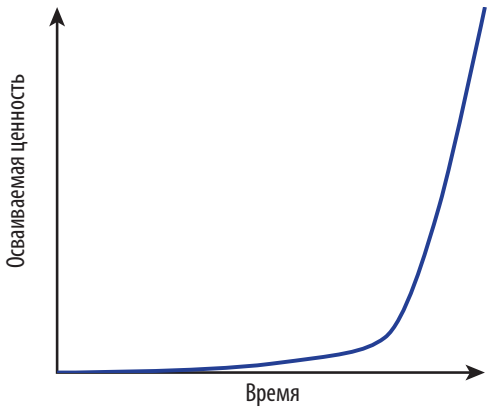


Рис. 7.20. Нереальный оптимистичный план

Такие нереалистичные, излишне оптимистичные планы обычно являются результатом обратного планирования. План даже может начинаться с самыми лучшими намерениями и следовать по критическому пути. К сожалению, вы вдруг обнаруживаете, что кто-то уже принял обязательства по проекту на конкретную дату без учета плана проекта или реальных возможностей команды. Вы берете оставшиеся активности, сваливаете их у предельного конечного срока и начинаете обратное планирование от этой точки. Только построение графика планируемой осваиваемой ценности позволит вам привлечь внимание к непрактичности этого плана и попытаться предотвратить неудачу. На рис. 7.21 изображен проект с таким поведением.

Аналогичным образом можно обнаруживать нереальные оптимистичные планы, такие как на рис. 7.22. Этот проект начинается хорошо, но затем результативность вдруг должна резко упасть — или, что более вероятно, проекту было выделено гораздо больше времени, чем требовалось. Проект на рис. 7.22 завершится неудачей, потому что он провоцирует украшательство и повышение сложности. По «здоровой» части кривой даже можно экстраполировать, когда проект должен был завершиться (где-то над перегибом кривой).

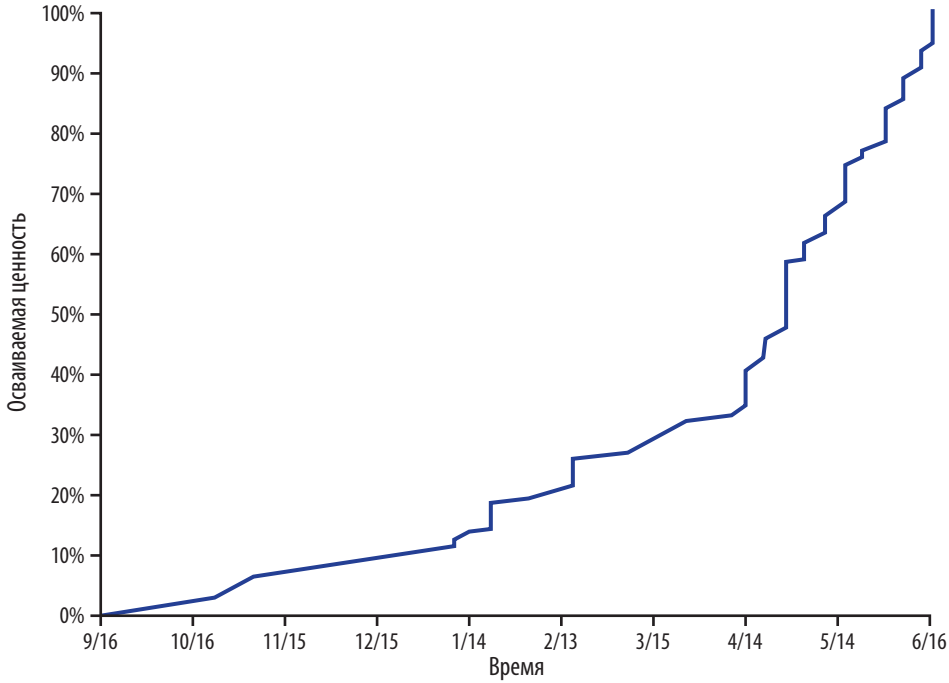


Рис. 7.21. Пример нереального оптимистичного плана

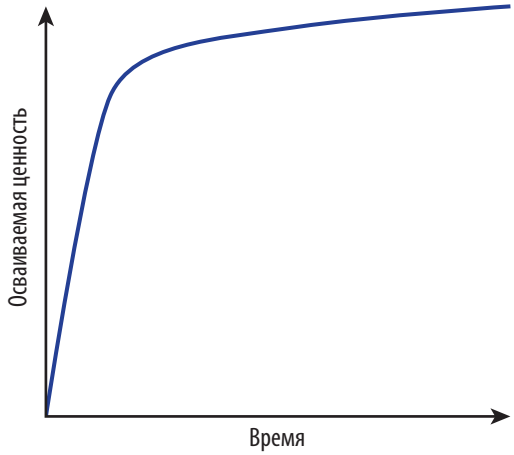


Рис. 7.22. Нереальный пессимистичный план

Пологая S-образная кривая

У проектов, использующих команду фиксированного размера, график планируемой осваиваемой ценности всегда представляет собой прямую линию. Как упоминалось ранее, размер команды не должен быть постоянным. У правильно укомплектованных и хорошо спланированных проектов график осваиваемой ценности всегда имеет вид пологой S-образной кривой (рис. 7.23).

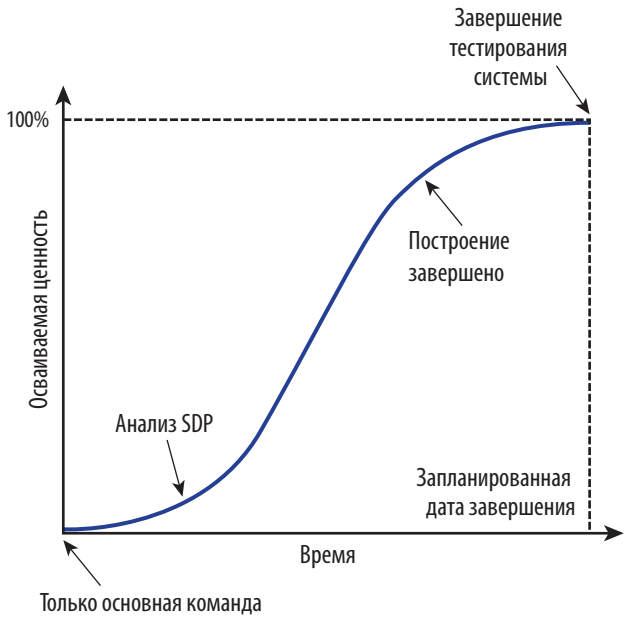


Рис. 7.23. Пологая S-образная кривая

Форма кривой планируемой осваиваемой ценности связана с планируемым распределением комплектования. В начале проекта доступна только основная команда, поэтому в начальной стадии сколько-нибудь заметная ценность не создается и кривая осваиваемой ценности почти горизонтальна. После анализа SDP в проект можно добавлять людей. С ростом размера команды также увеличивается ее результативность, так что кривая осваиваемой ценности становится все круче и круче. В определенный момент будет достигнута пиковая комплектация. В течение какого-то времени размер команды остается в основном фиксированным, поэтому график представляет собой прямую линию с максимальной результативностью в центре кривой. После того как вы начнете выводить ресурсы из проекта, кривая осваиваемой ценности выравнивается до завершения проекта. На рис. 7.24 изображена пологая S-образная кривая.

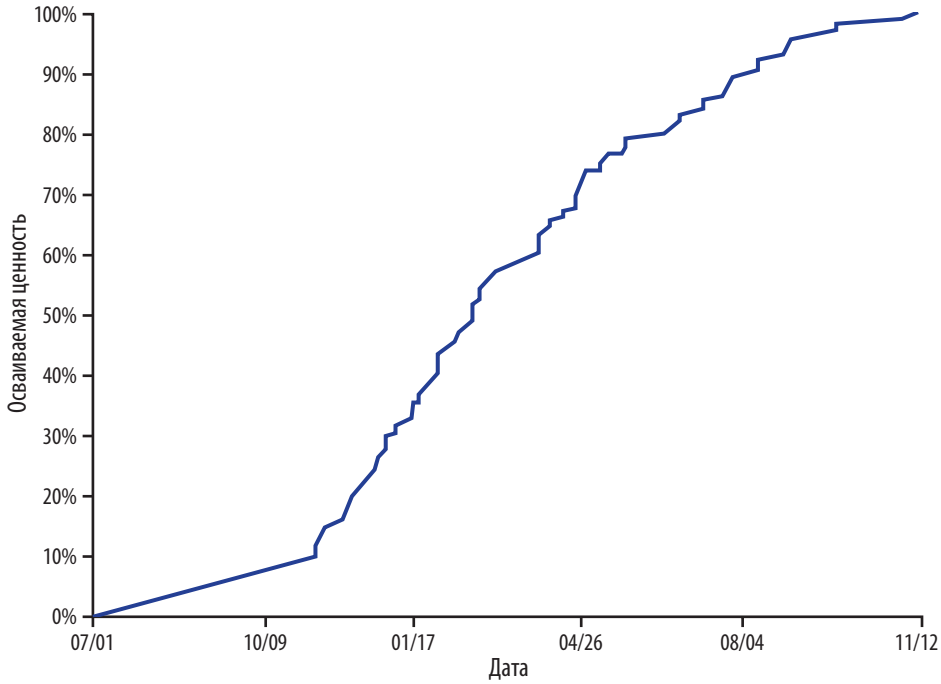


Рис. 7.24. Пример пологой S-образной кривой

ЛОГИСТИЧЕСКАЯ ФУНКЦИЯ

Пологая S-образная кривая планируемой осваиваемой ценности является специальным случаем логистической функции[1]. Обобщенная форма логистической функции может принимать любую S-образную форму (S, зеркальное S, перевернутое S), может охватывать любой диапазон значений и даже быть асимметричной.

Каждый процесс, в котором происходят изменения, может быть смоделирован при помощи логистической функции. Например, температура в комнате повышается и понижается в соответствии с логистической функцией — как и вес вашего тела, доля рынка, принадлежащая компании, радиоактивный распад, риск ожога кожи как функция расстояния от огня, статистические распределения, рост населения, эффективность проектирования, интеллект нейронных сетей и практически все остальное. Логистическая функция — самая важная из всех функций, известных человечеству, потому что она позволяет нам представлять в количественной форме и моделировать реальный мир с его динамической природой. Стандартная логистическая функция определяется следующим выражением:

[1] https://ru.wikipedia.org/wiki/Логистическая_функция

$$F(x) = \frac{1}{1 + e^{-x}}.$$

На рис. 7.25 изображена стандартная логистическая функция. Она асимптотически приближается к 0 и 1, пересекая ось y в точке $x = 0$ и $y = 0{,}5$. В последующих главах логистическая функция будет использоваться для моделирования рисков и сложности.

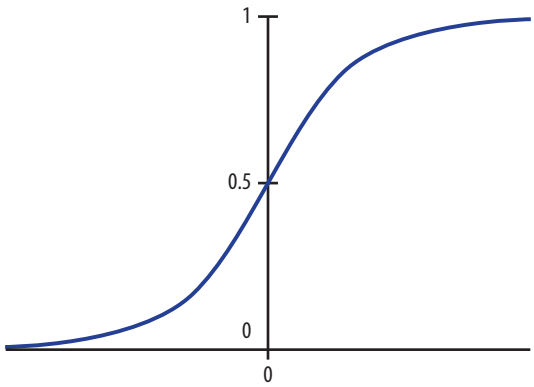


Рис. 7.25. Стандартная логистическая функция

Кривая осваиваемой ценности позволяет быстро и просто ответить на вопрос: «Насколько целесообразен этот план?» Если график планируемой осваиваемой ценности представляет собой прямую линию или на нем проявляются проблемы, показанные на рис. 7.20 и 7.22, проект в опасности. Если же он выглядит как пологая S-образная кривая, то по крайней мере можно надеяться на то, что план разумен и хорошо продуман.

Роли и обязанности

Задача архитектора — спроектировать систему и спланировать проект для построения этой системы. Скорее всего, архитектор является единственным участником команды, обладающим глубоким пониманием и представлением о правильной архитектуре, ограничениях технологии, зависимостях между активностями, проектных ограничениях как системы, так и проекта, а также относительных ресурсных навыках. Бесполезно ожидать, что высшее начальство, руководители проектов, руководители продуктов или разработчики смогут спроектировать проект. Всем им просто не хватает информации, понимания вопроса и квалификации, необходимой для планирования проекта. Кроме

того, планирование проектов не входит в их должностные обязанности. Однако архитектору понадобится информация, понимание и точка зрения менеджера проекта на стоимость ресурсов, сценарии доступности, предпосылки планирования, приоритеты и даже политические факторы. Менеджер продукта также играет исключительно важную роль в проработке архитектуры.

Архитектор планирует проект в ходе процесса, следующего за проектированием системы. Этот процесс идентичен тому, который используется во всех остальных инженерных дисциплинах: планирование проекта является частью инженерной деятельности. Проектировщик никогда не оставляет его рабочим, чтобы те как-нибудь разобрались сами на строительной площадке или на фабрике. Архитектор не несет ответственности за управление и контроль текущего состояния проекта. Вместо этого менеджер проекта назначает на проект рабочих и наблюдает за тем, как продвигается выполнение плана. Если в процессе выполнения ситуация изменится, руководитель проекта и архитектор должны совместными усилиями залатать брешь и перепланировать проект.

Понимание того, что архитектор должен планировать проект, является одним из факторов зрелости роли архитектора. Спрос на архитекторов появился в конце 1990-х годов как реакция на возрастающие эксплуатационные затраты и сложность программных систем. В наши дни архитекторы должны планировать системы, обеспечивающие удобство сопровождения, простоту повторного использования, расширяемость, масштабируемость, производительность, доступность, высокую скорость реакции и безопасность. Все эти факторы являются атрибутами планирования, и эти задачи должны решаться не на уровне технологии или ключевых слов, а на уровне правильности архитектуры.

Но этот список атрибутов планирования неполон. Эта глава начинается с определения успеха, и чтобы добиться успеха, необходимо добавить в список график, затраты и риски. Они являются атрибутами планирования в той же степени, что и остальные, и должны обеспечиваться правильным планированием проекта.

8 Сеть и временные резервы

Сетевой график является логическим представлением проекта для целей планирования. Метод анализа сети называется *методом критического пути*, хотя некритические активности для него так же важны, как и критические. Анализ критического пути прекрасно подходит для сложных проектов, от физического строительства до программных систем, и успешно применялся в течение многих лет. Посредством этого анализа можно найти продолжительность проекта и определить, где и когда следует назначать ресурсы.

Так как сетевой график проекта играет столь важную роль для планирования проекта, в этой главе подробно излагаются некоторые концепции, представленные в обзоре планирования проектов из главы 7. Многие приемы, термины и универсальные концепции, которые встретятся вам в этой главе, не зависят от специфики проекта и даже от отрасли. Идеи, представленные в этой короткой главе, закладывают основу для объективного и воспроизводимого анализа проекта. Два архитектора, анализирующие один сетевой проект, должны получить похожие результаты.

Сетевая диаграмма

Активностью в сетевом проекте называется любая задача, для решения которой требуется как время, так и ресурсы. Примеры активностей — разработка архитектуры, планирование проекта, построение сервисов, тестирование системы и даже курсы повышения квалификации. Проект представляет собой набор взаимосвязанных активностей, а сетевая диаграмма отражает эти активности и зависимости между ними. На сетевой диаграмме не существует понятия порядка выполнения или конкурентности задач.

Сетевые диаграммы часто намеренно не отображаются в масштабе, чтобы вы могли сосредоточиться исключительно на зависимостях и общей топологии сети. Отказ от масштаба в большинстве случаев также упрощает планирова-

ние проекта. Попытка выдержать масштаб на сетевой диаграмме также создает серьезные проблемы при изменении оценок, добавлении или удалении активностей, а также при перепланировании активностей.

Существуют два возможных представления сетевой диаграммы проекта: узловая диаграмма и стрелочная диаграмма (рис. 8.1).

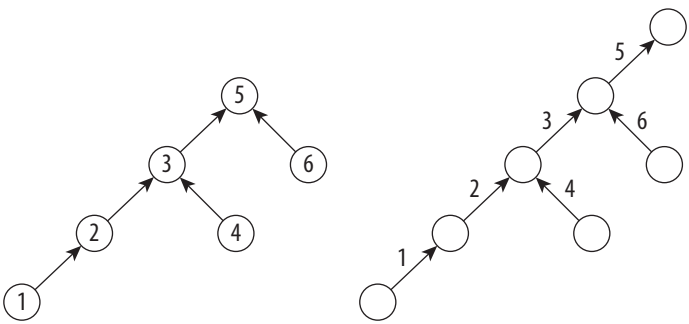


Рис. 8.1. Узловая диаграмма (слева) и эквивалентная стрелочная диаграмма (справа)

Узловая диаграмма

На узловой диаграмме каждый узел представляет активность. Например, в левой части рис. 8.1 каждый кружок представляет активность. Стрелки на узловой диаграмме представляют зависимости между активностями, а длина стрелки роли не играет. На стрелках время не тратится; все время тратится исключительно в узлах. Не существует простого способа отражения масштаба на узловых диаграммах, кроме увеличения радиуса узлов. Однако это приводит к загромождению диаграммы и затрудняет ее правильную интерпретацию.

Стрелочная диаграмма

На стрелочных диаграммах стрелки представляют активности, а узлы представляют зависимости между входными активностями, а также события, которые происходят при завершении входных активностей, как показано в правой части на рис. 8.1. Обратите внимание: обе диаграммы на рис. 8.1 представляют одну сеть, а два типа диаграмм эквивалентны (то есть любая сеть может быть представлена в любом из двух вариантов).

Так как узлы на стрелочной диаграмме представляют события, время никогда не тратится в узлах; иначе говоря, события происходят моментально. Как и на узловых диаграммах, время течет по направлению стрелок. Если вы захотите нарисовать стрелочную диаграмму в масштабе, следует сделать длину стрелок пропорциональной времени. Длина стрелки обычно не актуальна (в этой кни-

ге, если явно не указано обратное, все сетевые диаграммы изображаются без учета масштаба).

На стрелочной диаграмме все активности должны иметь начальное событие и событие завершения. Также рекомендуется добавить общее начальное событие и событие завершения для проекта в целом.

Фиктивные активности

Предположим, в сети на рис. 8.1 активность 4 также зависит от активности 1. Если активность 2 уже зависит от активности 1, на стрелочной диаграмме появляется проблема, потому что стрелка активности 1 не может расщепляться. Проблема решается введением фиктивной активности между событием завершения активности 1 и начальным событием активности 4 (обозначается пунктирной стрелкой на рис. 8.2). Фиктивная активность имеет нулевую продолжительность, а ее единственная цель — выражение зависимости от ее хвостового узла.

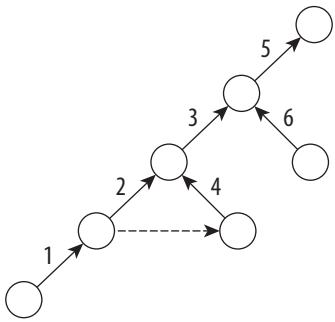


Рис. 8.2. Использование фиктивной активности

Стрелочные и узловые диаграммы

Хотя две разновидности диаграмм эквивалентны, у каждой есть свои достоинства и недостатки.

Один из доводов в пользу стрелочных диаграмм заключается в том, что события завершения становятся естественным местом для размещения *контрольных точек* (milestones). Контрольная точка — событие, обозначающее завершение важной части проекта. На узловых диаграммах в качестве контрольных точек обычно приходится добавлять активности нулевой продолжительности.

Практически любому человеку потребуется немного потренироваться, чтобы правильно рисовать и читать стрелочные диаграммы. С другой стороны, люди интуитивно рисуют и понимают узловые диаграммы, что выглядит очевидным преимуществом. На первый взгляд на узловых диаграммах не нужны фиктивные

активности, потому что вы можете добавить еще одну стрелку зависимости (например, еще одна стрелка между активностями 1 и 4 в левой части на рис. 8.1). По этим (несколько упрощенным) причинам подавляющее большинство инструментов для рисования сетевых диаграмм использует узловые диаграммы.

Напротив, по крайней мере четверо из заказчиков IDesign разработали средства для работы со стрелочными диаграммами (два из них включены в состав файлов, прилагаемых к книге). Они потратились на стрелочные диаграммы из-за одного критического недостатка всех узловых диаграмм. Взгляните на рис. 8.3.

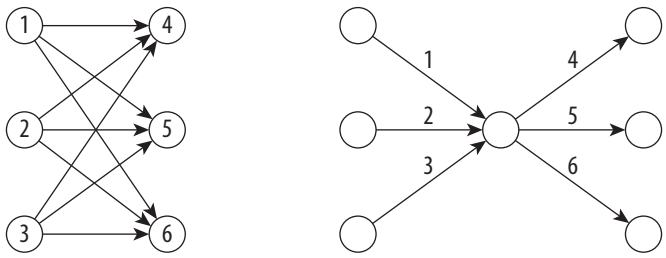


Рис. 8.3. Повторяющиеся зависимости на узловых и стрелочных диаграммах [адаптировано по материалам James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990)]

На рис. 8.3 изображены две идентичные сети, каждая из которых состоит из шести активностей: 1, 2, 3, 4, 5, 6. Активности 4, 5 и 6 зависят от активностей 1, 2 и 3. На стрелочной диаграмме сеть прямолинейна и понятна, тогда как соответствующая узловая диаграмма напоминает запутанный клубок. Узловую диаграмму можно немного упростить созданием фиктивного узла с нулевой продолжительностью, но его можно спутать с контрольной точкой.

Как выясняется, ситуация на рис. 8.3 весьма распространена в хорошо спроектированных системах с повторяющимися зависимостями на уровнях архитектуры. Например, активности 1, 2 и 3 могут быть сервисами *Доступ к ресурсу*, а активности 4, 5 и 6 могут быть *Менеджерами* и *Ядрами*, использующими все три сервиса *Доступ к ресурсу*. С узловыми диаграммами трудно разобраться в том, что происходит, даже в простом сетевом графике проекта вроде изображенного на рис. 8.3. А если добавить *Ресурсы*, *Клиенты* и *Вспомогательные средства*, диаграмма становится слишком сложной для понимания.

Бессмысленно рисовать сетевые диаграммы, которые никто не понимает. Главная цель сетевой диаграммы — передача информации: вы пытаетесь передать свой план проекта другим или даже самому себе. Наличие модели, которую никто не понимает и никто не поддерживает, противоречит исходной цели построения сетевой диаграммы.

ИСТОРИЯ МЕТОДА КРИТИЧЕСКОГО ПУТИ

Идеи сети активностей и критического пути как способа получить информацию о том, как строить проект, сколько времени это займет и сколько будет стоить, не новы. В строительной отрасли они успешно применялись десятилетиями. Метод критического пути был разработан компанией DuPont в составе «Манхэттенского проекта» в 1940-х годах[1], а также в ВМФ США в 1950-х годах в ходе разработки проекта ракет Polaris[2], запускаемых с подводных лодок. В обоих случаях анализ критического пути использовался для контролирования нарастающей сложности и решения других проблем, похожих на те, которые присущи крупным современным программным проектам. В 1959 году Джеймс Келли (James Kelley) опубликовал краткую статью[3] на основании опыта проектирования промышленных предприятий в компании DuPont. На первых восьми страницах статьи встречаются все знакомые элементы методологии: критический путь, стрелочные диаграммы, фиктивные активности, временные резервы и даже идеальная кривая «время-затраты».

В 1960-х годах в NASA метод критического пути использовался в качестве основного инструмента планирования для сокращения отставания и победы в лунной гонке.[4] Метод критического пути завоевал свою репутацию после роли, которую он сыграл в сносе сильно затянувшегося проекта Сиднейского оперного театра[5] и в обеспечении быстрого строительства Всемирного торгового центра в Нью-Йорке (самых высоких зданий в мире на то время); оба проекта были завершены в 1973 году.

Следовательно, вам стоит избегать узловых диаграмм и пользоваться стрелочными диаграммами. Время, потраченное на освоение стрелочных диаграмм, с избытком компенсируется компактной, четкой, чистой моделью вашего проекта. Отсутствие распространенных программ для построения стрелочных диаграмм, из-за чего стрелочные диаграммы приходится рисовать вручную, — это не всегда плохо. Рисование вручную полезно тем, что в процессе вы анализируете и проверяете зависимости активностей, что может даже открыть дополнительную информацию о проекте.

---

1 https://ru.wikipedia.org/wiki/Метод_критического_пути

2 https://ru.wikipedia.org/wiki/PERT#История

3 James E. Kelley and Morgan R. Walker, «Critical Path Planning and Scheduling,» Proceedings of the Eastern Joint Computer Conference, 1959

4 https://ntrs.nasa.gov/search.jsp?R=19760036633

5 James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990).

Временные резервы

Активности на критическом пути должны завершиться в запланированное время, чтобы избежать задержки всего проекта. Некритические события могут откладываться без нарушения графика; иначе говоря, они могут находиться в «плавающем» состоянии до своего начала. Проект, в котором нет никаких временных резервов, а все сетевые пути являются критическими, теоретически может выполнить свои обязательства, но на практике любой просчет может привести к задержке. С точки зрения планирования временной резерв формирует запас прочности проекта. При планировании проекта всегда следует ввести в сеть достаточный временной резерв. Группа разработки может расходовать его для компенсации непредвиденных задержек в некритических активностях. Проекты с низким временным резервом всегда сопряжены с высоким риском отставания от графика. Любые задержки в активностях с низким временным резервом, кроме самых мелких, приведут к тому, что активность станет критической и задержит реализацию проекта.

Обсуждение временных резервов до настоящего момента было несколько упрощенным, потому что на самом деле существует несколько разновидностей временных резервов. В этой главе обсуждаются две разновидности: общий временной резерв и свободный временной резерв.

Общий временной резерв

Общий временной резерв активности определяется временем, на которое можно задержать данную активность без задержки проекта в целом. Когда завершение активности откладывается на величину, меньшую ее общего временного резерва, это может привести к задержке последующих активностей, но срок завершения проекта при этом не изменится. Это означает, что общий временной резерв является аспектом цепочки активностей, а не конкретных активностей. Для примера возьмем сеть в верхней части рис. 8.4; критический путь выделен жирными линиями, а некритический путь (или цепочка активностей) располагается выше него.

Для целей нашего обсуждения на рис. 8.4 выдержан масштаб, так что длина каждой линии соответствует продолжительности каждой активности. Все некритические активности должны иметь одинаковый общий временной резерв, обозначенный пунктирной линией после стрелки активности. Представьте, что начало первой некритической активности в верхней половине диаграммы задерживается, так что на активность уходит больше времени, чем планировалось изначально. Во время выполнения этой активности задержка завершения активности поглощает общий временной резерв последующих активностей (как показано в нижней части диаграммы).

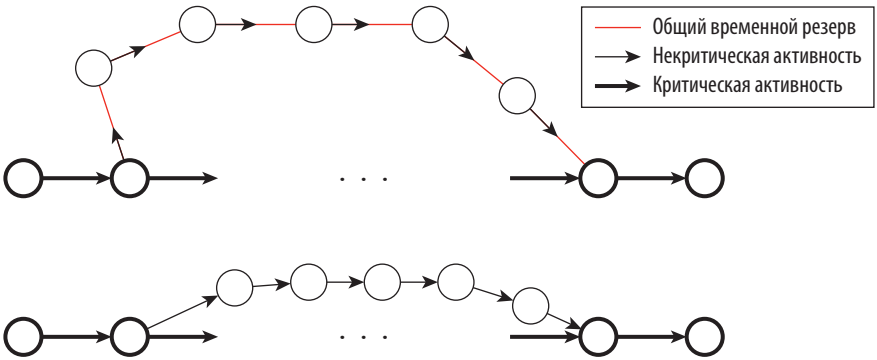


Рис. 8.4. Временной резерв как аспект цепочки активностей

Все некритические активности имеют некоторый временной резерв, а все активности одной некритической цепочки совместно используют часть общего временного резерва. Если активности также запланированы так, чтобы они запускались как можно раньше, все активности одной цепочки будут иметь одинаковое количество общего временного резерва. Потребление общего временного резерва где-то в начале цепочки отнимает его у последующих активностей, отчего те становятся более рискованными и критическими.

> **ПРИМЕЧАНИЕ** Как будет показано позднее в этой главе, общий временной резерв каждой активности в сети является ключевым фактором планирования проекта. В оставшейся части книги термином «временной резерв» всегда будет обозначаться общий временной резерв.

Свободный временной резерв

Свободный временной резерв активности определяет, на какое время можно задержать выполнение активности без последствий для всех остальных активностей в проекте. Если завершение активности задерживается на величину, меньшую либо равную ее свободному временному резерву, это никак не отразится на последующих активностях — и конечно, проект в целом не задержится. Возьмем рис. 8.5.

И снова для целей нашего обсуждения диаграмма на рис. 8.5 изображена в масштабе. Допустим, первая активность в некритической цепочке в верхней части диаграммы имеет свободный временной резерв, обозначенный пунктирной линией после стрелки активности. Представьте, что активность откладывается на период времени, меньший (или равный) его свободного временного резерва. Как видно из диаграммы, на последующих активностях эта задержка не отразится (нижняя часть диаграммы).

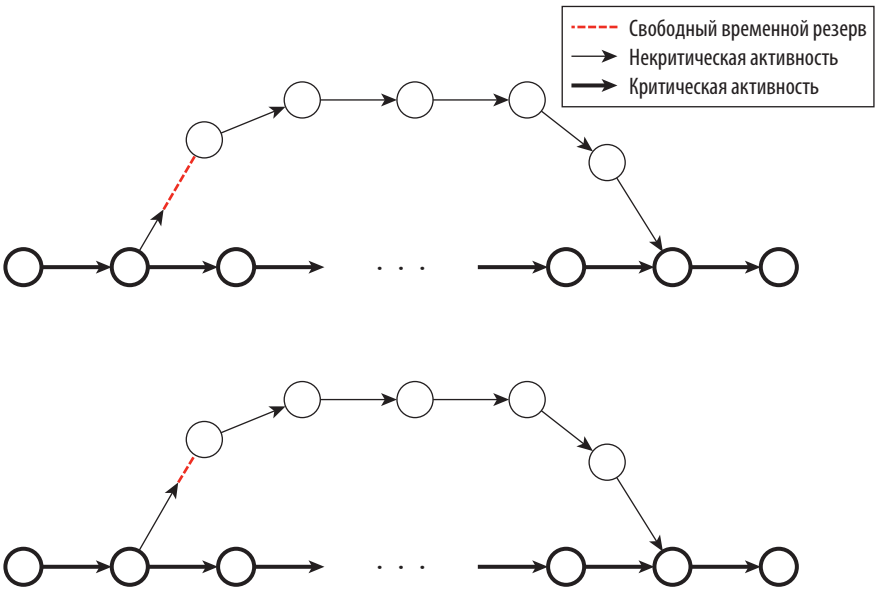


Рис. 8.5. Потребление свободного временного резерва

Интересно, что хотя любая некритическая активность всегда имеет некоторый общий временной резерв, свободного резерва у нее может и не быть. Если вы запланируете свои некритические активности так, чтобы они начинались как можно раньше, без свободных промежутков, то даже если эти активности являются некритическими, их свободный временной резерв равен 0, потому что любая задержка нарушит график выполнения других некритических активностей в цепочке. Тем не менее последняя активность в некритической цепочке, которая соединяется с критическим путем, всегда имеет некоторый свободный временной резерв (иначе она бы тоже была критической активностью).

Свободный временной резерв не приносит особой пользы в ходе планирования проекта, но он может оказаться очень полезным во время выполнения проекта. Если акт откладывается или превышает свою оценку объема работы, свободный временной резерв задерживаемой активности позволяет менеджеру проекта определить доступное время до того, как это затронет другие активности в проекте (если вообще затронет). Если задержка меньше свободного временного резерва задерживаемой активности, ничего делать не нужно. Если задержка превышает свободный временной резерв (но оказывается меньше общего временного резерва), менеджер проекта может вычесть свободный временной резерв из задержки, точно измерить степень влияния задержки на последующие активности и предпринять соответствующие действия.

Вычисление временного резерва

Временные резервы в сети проекта являются функцией продолжительностей активностей, их зависимостей и всех задержек, которые могут возникнуть в процессе выполнения. Они не имеют отношения к фактическим календарным датам, на которые запланированы эти активности. Тогда вы сможете вычислить временные резервы, даже если фактическая начальная дата проекта еще не определена.

В большинстве сетей сколько-нибудь серьезного размера ручное вычисление временных резервов создает высокий риск ошибок и быстро выходит из-под контроля, а результаты становятся бесполезными при любых изменениях в сети. К счастью, все эти вычисления являются чисто механическими, и для вычисления временных резервов следует использовать программы[1]. Зная общие временные резервы, вы сможете нанести их на сетевой график проекта, как показано на рис. 8.6. На диаграмме представлен пример сети проекта: черные числа — идентификаторы активностей, а числа под стрелками — общий временной резерв для некритических активностей.

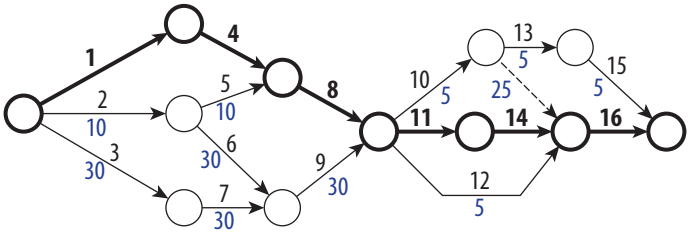


Рис. 8.6. Значения общего временного резерва на сетевой диаграмме

Хотя для планирования проекта требуется только общий временной резерв, вы также можете отметить свободный резерв на сетевой диаграмме. Эта информация очень сильно поможет менеджеру проекта в процессе выполнения.

Наглядное представление временных резервов

Способ представления временных резервов на сетевой диаграмме, использованный на рис. 8.6, не идеален. Люди медленно обрабатывают алфавитно-цифровые данные, и им трудно связать эту информацию с проектом. Вряд

[1] Для вычисления временного резерва каждой активности можно воспользоваться Microsoft Project: вставьте столбцы Total Slack и Free Slack, соответствующие общему и свободному временному резерву. О том, как вычислять временные резервы вручную, рассказано в книге James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990).

ли кому-нибудь удастся взглянуть на сложную сеть (или даже простую, как на рис. 8.6) и с ходу оценить критичность этой сети. Критичность сети указывает, где находятся области высокого риска и насколько проект близок к полностью критической сети. Общие временные резервы лучше наглядно представлять цветовым кодированием стрелок и узлов — например, использовать красный цвет для низких, желтый для средних и зеленый для высоких временных резервов. Значения можно разбить на диапазоны по нескольким критериям:

- *Относительная критичность*. Максимальное значение временного резерва всех активностей в сети делится на три равные части. Например, если максимальный временной резерв равен 45 дням, то красный диапазон лежит в границах от 1 до 15 дней, желтый — от 16 до 30 дней, а зеленый — от 31 до 45 дней. Этот способ хорошо работает при большом значений максимального временного резерва (например, более 30 дней), а значения временных резервов распределены равномерно.
- *Экспоненциальная критичность*. Относительная критичность подразумевает, что риск задержки более или менее равномерно распределен в диапазоне временных резервов. В действительности активность с 5 днями временного резерва с намного большей вероятностью приведет к нарушению графика, чем активность с 10 днями временного резерва, хотя оба значения могут относиться к красному диапазону по критерию относительной критичности. Для решения этой проблемы критерий экспоненциальной критичности делит диапазон максимального временного резерва на три неравных поддиапазона, размеры которых связаны экспоненциальной зависимостью. Я рекомендую выбрать точки деления на 1/9 и 1/3 диапазона: такие поддиапазоны имеют разумные размеры, но определяются более агрессивно, чем диапазоны с отношениями 1/4 и 1/2, а делители пропорциональны количеству цветов. Например, если максимальный временной резерв равен 45 дням, красный поддиапазон лежит в границах от 1 до 5 дней, желтый — от 6 до 15 дней, а зеленый — от 16 до 45 дней. Как и относительная критичность, экспоненциальная критичность хорошо работает при большом максимальном общем временном резерве (например, более 30 дней) и равномерном распределении временных резервов в диапазоне.
- *Абсолютная критичность*. Классификация по критерию абсолютной критичности не зависит ни от значения максимального временного резерва, ни от равномерности распределения временных резервов в диапазоне. Абсолютная критичность устанавливает абсолютный диапазон временного резерва для каждого поддиапазона цветовой классификации. Например, красные активности должны занимать от 1 до 9 дней, желтые — от 10 до 26 дней, и зеленые — от 27 дней. Классификация абсолютной критичности прямолинейна и хорошо работает в большинстве проектов. Ее недостаток заключается в том, что она может требовать настройки диапазонов для

конкретных проектов в соответствии с рисками. Например, временной резерв в 10 дней может относиться к зеленому диапазону в 2-месячном проекте, но к красному диапазону в проекте продолжительностью в один год.

На рис. 8.7 изображена та же сеть, что и на рис. 8.6, с цветовым кодированием по критерию абсолютной критичности и предложенными значениями диапазонов. Критические активности, выделенные черным цветом, не имеют временного резерва.

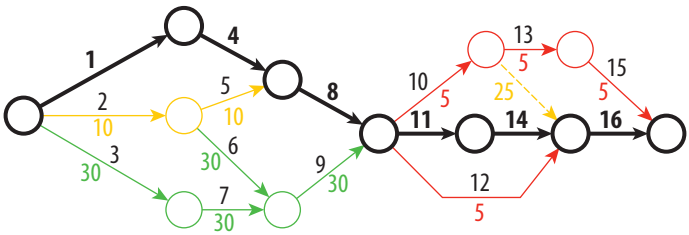


Рис. 8.7. Цветовое кодирование временного резерва

Сравните, насколько проще интерпретируется визуальная информация на рис. 8.7 по сравнению с той же текстовой информацией на рис. 8.6. Вы немедленно видите, что вторая часть проекта сопряжена с высоким риском.

Активное управление проектами

Многие компетентные руководители проектов активно управляют своими проектами на критическом пути. Так как любая задержка на этом пути приведет к нарушению сроков, менеджер проекта зорко следит за критическим путем. Если хорошо управляемые проекты все равно нарушают сроки, такие задержки обычно происходят не из-за недостаточного внимания к критическим активностям. Мой опыт показывает, что основная причина для неудач в хорошо управляемых проектах — превращение некритичных активностей в критичные. Обычно это происходит из-за того, что исходные некритические активности не получили ресурсов по запланированному графику, это привело к их выходу за пределы общего временного резерва, они стали критическими и задержали реализацию проекта.

Чтобы не быть захваченным врасплох некритическими активностями, менеджер проекта должен активно отслеживать общий временной резерв по всем цепочкам некритических активностей. Менеджер проекта может регулярно вычислять общий временной резерв для каждой цепочки и даже экстраполировать проявляющиеся тенденции, чтобы понять, в какой точке она станет критической. Менеджер проекта должен отслеживать цепочки с относительно

высокой частотой (например, еженедельно), потому что временной резерв цепочки часто ведет себя как ступенчатая функция с нелинейным падением из-за зависимостей от других активностей или ресурсов.

Составление графика на основе временных резервов

Как упоминалось в главе 7, самый безопасный и эффективный способ назначения ресурсов между активностями основан на временных резервах, или, с учетом определения из этой главы, общих временных резервах. Это самый безопасный способ, потому что сначала вы разбираетесь с более рискованными активностями, и самый эффективный, потому что он доводит до максимума процент времени, в течение которого используются ресурсы.

Рассмотрим диаграмму планирования, показанную на рис. 8.8. Здесь длина каждой цветной полосы представляет продолжительность активности, масштабированную по времени, а позиция левого или правого края выравнивается в соответствии с графиком.

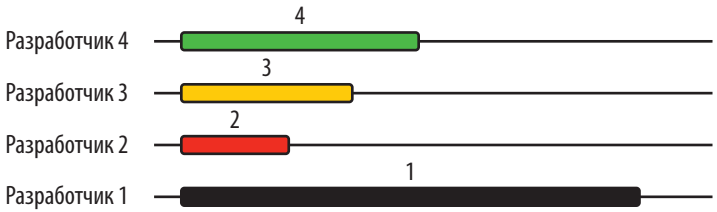


Рис. 8.8. Потребление свободного временного резерва

На диаграмме изображены четыре активности: 1, 2, 3, 4. Все активности готовы стартовать одновременно. Из-за последующих активностей (не показаны на диаграмме) активность 1 критична, хотя активности 2, 3 и 4 имеют разные уровни общего временного резерва, обозначенного цветом: 2 — красный (низкий временной резерв), 3 — желтый (средний временной резерв), 4 — зеленый (высокий временной резерв). Предположим, все эти активности разработки могут одинаково хорошо выполняться всеми разработчиками и проблем с непрерывностью задач не существует. При комплектовании проекта сначала необходимо назначить разработчика на критическую активность 1. Если у вас имеется второй разработчик, он назначается на активность 2, которая имеет наименьший временной резерв среди всех остальных активностей. Такая схема позволяет использовать до четырех разработчиков, которые начинают работать над каждой из активностей сразу же, когда появится такая возможность.

Также возможно укомплектовать проект всего двумя разработчиками (рис. 8.9). Как и прежде, первый разработчик трудится над активностью 1. Второй разработчик начинает работать над активностью 2 сразу же, когда это становится возможным, потому что нет смысла откладывать околокритическую активность и делать ее критической.

Как только активность 2 будет завершена, второй разработчик переходит к оставшейся активности с наименьшим временным резервом (активность 3). Это требует перепланирования 3 по временной шкале, пока второй разработчик не станет доступным после завершения активности 2. Это возможно только за счет потребления (сокращения) временного резерва активности 3; из-за каких-то последующих зависимостей в данном примере это приводит к тому, что активность 3 переходит в красную категорию. После того как активность 3 будет завершена, второй разработчик переходит к работе над активностью 4. Это тоже становится возможным только за счет потребления доступного временного резерва активности 4; хотя в данном примере это приемлемо, временной резерв активности 4 переходит из зеленой категории в желтую.

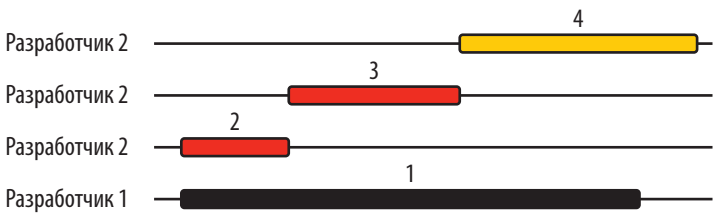


Рис. 8.9. Замена ресурсов временным резервом

В данной форме комплектования временной резерв заменяет ресурсы — и фактически затраты. При назначении ресурсов, доступных активностям, временной резерв может использоваться одним из двух способов: либо ресурсы назначаются на основании временных резервов, от низких к высоким, либо при необходимости временной резерв активностей потребляется для комплектования проектов с наименьшим уровнем ресурсов без задержки проекта.

> **ПРИМЕЧАНИЕ** Смещение активности 3 по временной шкале до освобождения разработчика, работающего над активностью 2, равносильно тому, что активность 3 зависит от активности 2. Это хороший способ модификации сети, отражающей зависимость от ресурса. Вспомните, о чем говорилось в главе 7: сетевая диаграмма является не просто сетью активностей, а сетью зависимостей, а зависимости ресурсов в первую очередь являются зависимостями.

Временной резерв и риск

Как уже было сказано, назначение ресурсов на основании временного резерва позволяет расходовать временной резерв вместо затрат. Возможно, у вас появится искушение потратить весь временной резерв проекта для снижения затрат, но так поступать не следует, потому что проект с недостаточным временным резервом менее устойчив к задержкам. Заменяя ресурсы временным резервом, вы сокращаете затраты, но повышаете риск. Фактически вы не только обмениваете ресурсы на временной резерв, но и сокращаете затраты за счет повышения риска. Таким образом, расходование временного резерва является трехсторонней операцией. В примере на рис. 8.9 использование двух разработчиков вместо четырех приводит к сокращению затрат, но при этом появляется побочный эффект: проект становится более рискованным. В ходе планирования проекта необходимо постоянно управлять оставшимся временным резервом, тем самым управляя рисками проекта. Это позволит вам определить несколько вариантов с разными комбинациями сроков, затрат и рисков.

9 Время и затраты

Самый быстрый способ реализации любой системы — построение ее по критическому пути. Хорошо спроектированный проект также эффективно распределяет минимально необходимые ресурсы по критическому пути, но продолжительность реализации проекта по-прежнему ограничивается его критическим путем. Выполнение можно ускорить за счет применения инженерных практик, обеспечивающих быструю и чистую разработку. Кроме этих практик разработки, в этой главе рассматриваются возможности сокращения сроков разработки за счет сжатия критического пути. Основной способ сокращения сроков — переработка проекта с определением нескольких более коротких планов проекта с еще большей степенью сжатия. После этого проявляется фундаментальная концепция кривой «время-затраты» и взаимодействие времени и затрат в проекте. В результате вы получаете набор вариантов планирования проекта, которые позволяют приспособиться к изначальным пожеланиям руководства относительно времени и затрат, а также быстро адаптироваться к возможным непредвиденным изменениям.

Ускорение программных проектов

Вопреки распространенному мнению, для соблюдения сроков недостаточно просто больше работать или назначить в проект больше людей. Для этого нужно работать умнее, четче и правильнее, применяя профессиональный опыт. В общем случае в любом программном проекте для ускорения проекта в целом могут применяться следующие методы:

- *Обеспечение качества*. В большинстве команд операции контроля качества и тестирования ошибочно причисляются к области обеспечения качества, или QA (Quality Assurance). Истинная сфера QA не имеет отношения к тестированию. Как правило, в ней задействуется опытный эксперт, который отвечает на вопрос: что потребуется для обеспечения качества? В его ответе должно быть указано, как ориентировать весь процесс разработки для

контроля качества, как предотвратить само возникновение проблем, как обнаружить и исправить корневые причины проблем. Само присутствие QA-специалиста — признак организационной зрелости, который почти всегда свидетельствует о приверженности качеству, понимании того, что качество не образуется само по себе, и подтверждении того, что организация должна активно к нему стремиться. QA-специалист иногда отвечает за планирование процесса и проработку ключевых фаз. Поскольку качество ведет к продуктивности, правильные меры обеспечения качества всегда ускоряют работу по графику и выделяют организации, практикующие QA, из остальных представителей отрасли.

- *Привлечение инженеров по тестированию.* Инженеры по тестированию — не рядовые тестировщики, а полноценные программисты, которые проектируют и пишут код, предназначенный для нарушения работоспособности кода системы. Как правило, инженеры по тестированию обладают более высокой технической квалификацией, чем обычные разработчики, потому что написание тестового кода часто включает более трудные задачи: разработку фиктивных каналов коммуникаций; проектирование и разработку регрессионного тестирования; проектирование тестовых оснасток, имитаторов, средств автоматизации и т. д. Инженеры по тестированию до мелочей знают архитектуру и внутреннее устройство системы и пользуются этими знаниями для того, чтобы нарушать работу системы на каждом шаге. Наличие такой «антисистемной» системы, готовой развалить ваш продукт, творит чудеса для качества, потому что вы можете выявлять проблемы сразу же после их возникновения, изолировать корневые причины, избегать каскадных эффектов при изменениях, устранять суперпозиции дефектов, маскирующих другие дефекты, и значительно сокращать цикл решения проблем. Ничто не ускоряет работу по графику в такой степени, как постоянная, свободная от дефектов кодовая база.

- *Добавление тестировщиков.* Как правило, в командах разработчиков больше, чем тестировщиков. В проектах, в которых недостаточно тестировщиков, один-два тестировщика не могут угнаться за командой, и их участие часто ограничивается проведением тестирования с незначительной полезностью. Такое тестирование слишком монотонно, оно не изменяется в зависимости от размера команды или растущей сложности системы, а система часто рассматривается как «черный ящик». Это не означает, что качественное тестирование при этом не проводится, это означает, что скорее его основная часть перекладывается на разработчиков. Изменение пропорции тестировщиков к разработчикам до 1:1 или даже 2:1 (в пользу тестировщиков) позволяет разработчикам проводить меньше времени за тестированием и больше — за созданием непосредственной пользы для проекта.

- *Вложения в инфраструктуру.* Всем программным системам требуются такие общие средства, как средства безопасности, очереди сообщений и шина сообщений, размещение, публикация событий, ведение журнала, инстру-

ментальные средства, диагностика и профилирование, а также регрессионное тестирование и автоматизация тестирования. Многим программным системам требуются средства управления конфигурацией, сценарии развертывания, процесс сборки, ежедневные сборки и тесты состояния (все это часто объединяется в категорию DevOps). Вместо того чтобы заставлять каждого разработчика писать собственную уникальную инфраструктуру, следует вложиться в построение (и сопровождение) инфраструктурного каркаса для всей команды, который будет решать большинство перечисленных задач. Это позволяет разработчикам сосредоточиться на задачах программирования, имеющих прямую коммерческую ценность, обеспечивает экономию за счет масштаба, помогает новым разработчикам быстрее войти в курс дела, снижает уровень стресса и трения в команде, а также сокращает время разработки системы.

- *Повышение квалификации разработки*. Для современных сред характерен очень высокий темп изменений. Многие разработчики не могут угнаться за новейшими языками, инструментами, фреймворками, облачными платформами и другими новшествами. Даже самые лучшие разработчики вечно осваивают новые технологии, и они проводят непропорциональное время за неструктурированными, хаотичными поисками. Что еще хуже, некоторые разработчики настолько не справляются с нагрузкой, что занимаются копированием кода из интернета без реального понимания краткосрочных и долгосрочных последствий (включая юридические) своих действий. Чтобы справиться с этой проблемой, следует выделять время и ресурсы для обучения разработчиков технологиям, методологиям и имеющимся инструментам. Наличие компетентных разработчиков ускорит разработку любого программного продукта.

- *Совершенствование процесса*. Многие среды разработки страдают от ограниченности процесса. Они выполняют все положенные действия как ритуал, но не имеют четкого понимания или правильного восприятия, лежащего в основе активностей. Активности, лишенные реального содержания, не приносят никакой реальной пользы и часто только ухудшают ситуацию по принципу карго-культа[1]. О процессах разработки программных продуктов были написаны целые тома. Изучайте проверенные передовые практики и разработайте план совершенствования, который решит проблемы качества, сроков и бюджета. Отсортируйте методы, входящие в план совершенствования, по результативности и простоте принятия и заранее анализируйте причины, по которым они не применялись ранее. Сформулируйте стандартные оперативные процедуры, настаивайте на том, чтобы команда и вы сами следовали этим процедурам, и даже заставляйте при необходимости. Со временем ваши проекты станут более предсказуемыми, и вы сможете реализовывать их в соответствии с заданным графиком.

[1] https://ru.wikipedia.org/wiki/Карго-культ

- *Адаптация и применение стандартов.* В подробном стандарте программирования определяются правила формирования имен и стиль, практики программирования, настройки и структура проекта, ваши собственные правила, регламенты вашей команды и известные потенциальные проблемы. Стандарт помогает обеспечить применение передовых практик разработки и избегать ошибок, чтобы новички могли быстро подняться до уровня ветеранов. Код становится более единообразным, и пропадают те проблемы, которые часто встречаются при работе над кодом, написанным другим разработчиком. Соблюдение стандартов повышает вероятность успеха и сокращает время, которое могло бы уйти на разработку системы.
- *Доступность для внутренних экспертов.* В большинстве команд вы не найдете экспертов мирового уровня. Задача команды — понять суть бизнеса и создать систему, а не продемонстрировать свою квалификацию в безопасности, размещении, UX, облачных технологиях, AI, BI, больших данных или архитектуре баз данных. На эти попытки «изобретения велосипеда» тратится много времени, и они никогда не бывают такими же эффективными, как использование доступных и проверенных знаний (вспомните проблему 2% из главы 2). Гораздо лучше и быстрее довериться внутренним экспертам. Используйте этих экспертов там, где это потребуется, — это позволит избежать дорогостоящих ошибок.
- *Проведение партнерского рецензирования.* Лучший отладчик — человеческий глаз. Разработчики часто обнаруживают проблемы в коде коллег намного быстрее, чем потребуется для диагностики и исправления проблем после того, как код станет частью системы. Этот принцип также справедлив для дефектов требований, для плана проектирования и плана тестирования для каждого из сервисов в системе. Команда должна проанализировать все эти аспекты, чтобы обеспечить наивысшее качество кодовой базы.

> **ПРИМЕЧАНИЕ** Отрасль разработки программного обеспечения настолько хаотична, что эти практики на уровне здравого смысла могут показаться чужеродными многим современным разработчикам. Тем не менее если вы проигнорируете их и будете делать то же, что и раньше, только в большем объеме, это не приведет к ускорению проекта. Действия, породившие проблему, не могут использоваться для ее решения. Со временем применение этих практик (или хотя бы их части) повысит производительность вашей команды, ее приверженность успеху и способность уверенно браться за сложные планы.

Передовые практики программирования ускоряют проект в целом независимо от конкретных активностей или сетевого графика самого проекта. Они эффективны в любом проекте, в любой среде и при использовании любых технологий. Хотя такой способ улучшения проекта может показаться дорогостоящим, в конечном итоге он может привести к экономии. Сокращение времени, затраченного на разработку системы, компенсирует дополнительные затраты на ее улучшение.

Уплотнение графика

К сожалению, ни один из элементов приведенного выше списка методов ускорения не гарантирует быстрого решения проблем с графиком; всем им требуется время, чтобы проявить свою эффективность. Тем не менее существует пара возможностей для немедленного ускорения работы — вы должны либо работать с лучшими ресурсами, либо поискать возможности выполнения параллельной работы. Применение этих приемов позволяет вам уплотнить график проекта. Уплотнение графика вовсе не означает, что та же работа будет выполняться быстрее. Речь идет о том, что вы будете достигать тех же целей быстрее, причем часто для более быстрого завершения задачи или проекта придется выполнить больший объем работы. Эти два способа уплотнения могут использоваться в сочетании друг с другом или по отдельности, в отдельных частях проекта, в проекте в целом или на уровне отдельных активностей. Оба способа уплотнения в конечном итоге повышают прямые затраты проекта (определение приводится далее) при сокращении сроков.

Использование лучших ресурсов

Старшие разработчики реализуют свою часть системы быстрее, чем младшие. Тем не менее существует распространенное заблуждение, будто это различие объясняется тем, что они быстрее программируют. Часто младшие разработчики программируют намного быстрее, чем старшие. Старшие разработчики проводят за программированием минимально возможную часть своего времени, а основное время проводится за проектированием программного модуля, взаимодействий и методов, которые они собираются использовать при тестировании. Старшие разработчики пишут тестовые оснастки, имитаторы и эмуляторы для компонентов, над которыми они работают, и для потребляемых ими сервисов. Они документируют свою работу, предвидят последствия каждого решения относительно программирования, учитывают удобство сопровождения и расширяемость своих сервисов, а также такие аспекты, как безопасность. Следовательно, хотя на единицу времени старшие разработчики пишут код медленнее, чем младшие, они быстрее справляются со своими задачами. Как нетрудно догадаться, старшие разработчики пользуются высоким спросом и запрашивают более высокую оплату, чем младшие. Эти лучшие ресурсы должны назначаться только на критические активности, поскольку их использование за пределами критического пути не повлияет на график.

Параллельная работа

Как правило, если вы берете последовательный набор активностей и находите возможность выполнять эти активности параллельно, это приводит к сокращению графика. Существуют два возможных способа параллельной работы.

В первом случае вы извлекаете внутренние фазы активности и перемещаете их в другую точку проекта. Во втором случае вы удаляете зависимости между активностями, чтобы работать над этими активностями параллельно (как объяснялось в главе 7, одновременное назначение нескольких людей на одну активность не работает).

Расщепление активностей

Вместо того чтобы выполнять внутренние фазы активности последовательно, можно расщепить активность. Некоторые из менее зависимых фаз планируются параллельно с другими активностями проекта, до или после активности. Среди хороших кандидатов для внутренних фаз, перемещаемых к началу проекта (то есть до остатка активности), можно выделить подробное проектирование, документирование, построение эмуляторов, план тестирования сервисов, тестовую оснастку, проектирование API, UI-проектирование и т. д. Кандидаты для внутренних фаз, перемещаемых к концу проекта, — интеграция с другими сервисами, модульное тестирование и повторное документирование. Расщепление активности сокращает время, занимаемое ею на критическом пути, и сокращает продолжительность работы над проектом.

Удаление зависимостей

Вместо того чтобы последовательно работать над зависимыми активностями, стоит поискать возможности сокращения и даже устранения зависимостей между активностями и организации параллельной работы над ними. Если в проекте присутствует активность A, зависящая от активности B, которая, в свою очередь, зависит от активности C, продолжительность проекта будет определяться суммой продолжительностей этих трех активностей. Но если вам удастся удалить зависимость между A и B, вы сможете работать над A параллельно с B и C, что приведет к соответствующему уплотнению графика.

Удаление зависимостей часто требует вложений в дополнительные активности, которые делают возможной параллельную работу:

- *Проектирование контракта.* Создав отдельную активность для проектирования контракта сервиса, вы можете предоставить интерфейс или контракт потребителям, а потом начать работать над ними еще до завершения сервиса, от которого они зависят. Возможно, предоставление контракта не снимет зависимость полностью, но откроет возможность для выполнения параллельной работы до определенного уровня. То же относится к проектированию UI, сообщений, API и протоколов между подсистемами и даже системами.
- *Разработка эмуляторов.* Для спроектированного контракта можно написать простой сервис, который эмулирует реальный сервис. Такая реализация может быть очень простой (например, всегда возвращать одни и те же

результаты без ошибок), что приведет к дальнейшему устранению зависимостей.

- *Разработка имитаторов.* Вместо простого эмулятора можно разработать полноценный имитатор сервиса или группы сервисов. Имитатор может поддерживать состояние, внедрять ошибки и обладать поведением, неотличимым от поведения реального сервиса. Иногда написание хорошего имитатора может быть делом более сложным, чем построение реального сервиса. Тем не менее имитатор разрывает зависимости между сервисом и его клиентами, позволяя реализовать более высокую степень параллелизма в работе.
- *Повторная интеграция и тестирование.* Даже если у вас есть отличный имитатор для сервиса, клиент, разработанный для этого имитатора, должен быть причиной для беспокойства. После того как реальный сервис будет завершен, вы должны повторить интеграцию и тестирование этого сервиса со всеми клиентами, разработанными для имитатора.

Кандидаты для параллельной работы

Иногда лучшие кандидаты для параллельной работы очевидны из диаграммы распределения кадров. Если диаграмма содержит несколько импульсов, возможно, эти импульсы можно разделить.

Взгляните на диаграмму на рис. 9.1: на ней хорошо видны три импульса. В исходном плане все три выполнялись последовательно из-за зависимостей между выходом каждого импульса, который становился входом для следующего импульса. Если вам удастся каким-то образом удалить эти зависимости, вы сможете работать над одним или двумя импульсами параллельно, что приведет к значительному уплотнению графика.

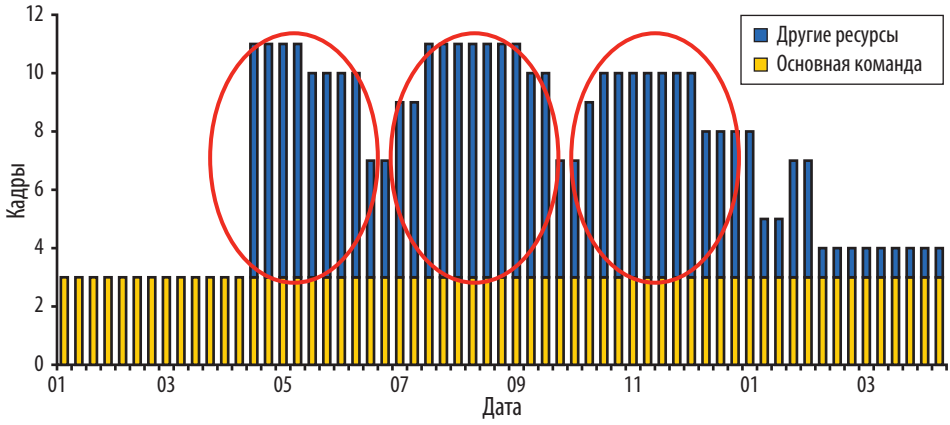


Рис. 9.1. Кандидаты для параллельной работы

Параллельная работа и затраты

Обе формы параллельной работы — расщепление активностей и удаление зависимостей между активностями — часто требуют дополнительных ресурсов. Для выполнения извлеченных фаз параллельно с другими активностями проекту часто требуется больше ресурсов. Проекту также потребуются дополнительные ресурсы для работы над дополнительными активностями, обеспечивающими параллельную работу, например дополнительными разработчиками для повторной интеграции и дополнительными тестировщиками для повторного тестирования. Это повышает стоимость проекта и рабочую нагрузку. В частности, дополнительные ресурсы приводят к увеличению размера команды, увеличению пикового размера команды, повышению шума и меньшей эффективности выполнения. Снижение эффективности обернется еще большим повышением затрат, потому что вы будете получать меньше от каждого участника команды.

Существующая команда может быть не способна к параллельной работе по разным причинам (отсутствие архитектора, отсутствие старших разработчиков, неподходящий размер команды), и вам придется пользоваться услугами дорогостоящих внешних специалистов. Даже если вы можете себе позволить общую стоимость проекта, параллельная работа повысит скорость оборота денежных средств, а проект может стать неприемлемым. Короче говоря, параллельная работа не бесплатна.

Опасности параллельной работы

Обычно удаление зависимостей между активностями сродни обезвреживанию мины — действовать нужно очень осторожно. Параллельная работа часто увеличивает сложность исполнения проекта, а это заметно повышает требования к менеджеру проекта, ответственному за проект. Прежде чем браться за параллельную работу, стоит заняться инфраструктурой, которая бы ускоряла все активности в проекте без изменения зависимостей между активностями. Пожалуй, это проще и безопаснее параллельной работы.

Несмотря на все сказанное, параллельная работа сократит время выхода на рынок. Выбирая решение об уплотненном параллельном режиме, тщательно взвесьте риски и затраты на параллельное выполнение с ожидаемым сокращением графика.

Кривая зависимости затрат от времени

По крайней мере изначально повышение затрат позволяет быстрее завершить работу над любым проектом. В большинстве проектов зависимость между временем и затратами нелинейна, но в идеале она напоминает кривую на рис. 9.2.

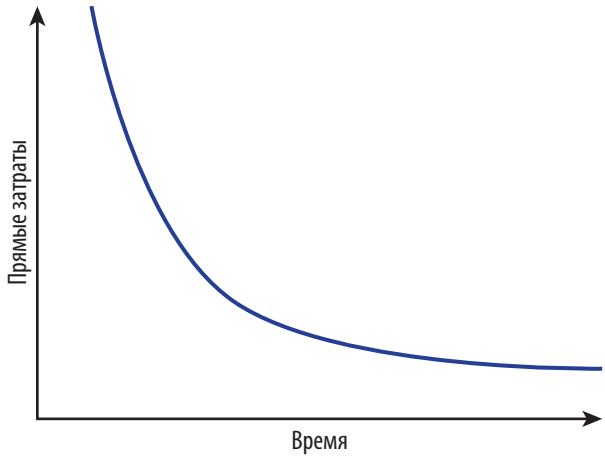


Рис. 9.2. Идеальная кривая «время-затраты»

Например, представьте проект на 10 человеко-лет, который состоит исключительно из активностей, связанных с программированием. Если поручить этот проект одному разработчику, на его выполнение уйдет 10 лет. С другой стороны, с двумя разработчиками тот же проект с большой вероятностью займет 7 и более лет, а не 5. Чтобы завершить проект за 5 лет, с большой вероятностью потребуются как минимум 3 разработчика, а скорее 5 или даже 6. Эти затраты (10 лет для 10 человеко-лет затрат, 7 лет для 14 человеко-лет, 5 лет для 30 человеко-лет) действительно выражают нелинейную зависимость затрат от времени.

Точки на кривой «время-затраты»

Кривая «время-затраты», изображенная на рис. 9.2, идеальна и нереалистична. Она предполагает, что при достаточно большом бюджете проект может быть реализован практически мгновенно. Здравый смысл подсказывает, что это предположение ошибочно. Например, ни при каком финансировании не удастся завершить проект на 10 человеко-месяцев за месяц (или хотя бы за год). У любых усилий по уплотнению графика есть естественный предел. Аналогичным образом кривая «время-затраты» на рис. 9.2 показывает, что при увеличении времени затраты на проект уменьшаются, тогда как выделение проектам времени большего, чем реально необходимо, приводит к повышению их затрат (как обсуждалось в главе 7).

Хотя кривая «время-затраты» на рис. 9.2 неверна, мы можем обсудить некоторые ее аспекты, встречающиеся во всех проектах. Эти аспекты являются результатом нескольких классических предположений из области планирования. На рис. 9.3 изображена реальная кривая «время-затраты».

Рис. 9.3. Реальная кривая «время-затраты» [адаптировано по материалам James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990)]

Нормальное решение

Проект всегда можно спроектировать в предположении о том, что вы располагаете неограниченными ресурсами и что каждый ресурс будет доступен тогда, когда он потребуется. В то же время проект следует планировать с учетом минимальных затрат и по возможности стараться не запрашивать больше ресурсов, чем действительно необходимо. Как объяснялось в главе 7, вы можете вычислить наименьший уровень ресурсов, который позволит беспрепятственно перемещаться по критическому пути. Вы получите наименее затратный способ построения системы и формирования самой эффективной команды. Такой вариант плана проекта называется *нормальным решением*. Нормальное решение представляет наименее ограниченный, или *естественный*, способ построения системы.

Неэкономичная зона

Предположим, продолжительность нормального решения проекта составляет один год. Если на этот же проект выделяется более одного года, он всегда будет стоить дороже. Дополнительные затраты происходят от блокировки ресурсов на более длительные периоды времени, от накапливаемых лишних затрат, от украшательства, от повышения сложности, а также от сокращения вероятно-

сти успеха. Таким образом, точки справа от нормального решения на кривой «время-затраты» относятся к неэкономичной зоне проекта.

Уплотненные решения

Нормальное решение можно уплотнить при помощи методов, описанных ранее в этой главе. Хотя все полученные уплотненные решения имеют меньшую продолжительность, они также обходятся дороже — скорее всего, в нелинейной зависимости.

Очевидно, все усилия по уплотнению должны быть направлены только на активности на критическом пути, потому что уплотнение некритических активностей никак не отражается на графике. Все уплотненные решения располагаются слева от нормального решения на кривой «время-затраты».

Минимальная продолжительность решения

По мере сжатия проекта затраты растут. В какой-то момент критический путь будет полностью уплотнен, потому что других кандидатов для параллельной работы нет, а вы уже назначили своих лучших людей на критические активности. При достижении этой точки вы получаете решение с минимальным временем (продолжительностью). Каждый проект всегда имеет точку минимальной продолжительности, в которой никакое количество денег, усилий или воли уже не ускорит его реализацию.

Решение с полным уплотнением

Невозможно построить проект за срок, меньший его наименьшей возможной продолжительности, но потерять деньги можно всегда. Ничто не мешает вам уплотнить все активности в проекте — как критические, так и некритические. Этот проект не будет завершен быстрее минимальной продолжительности, но безусловно обойдется дороже. Эта точка на кривой «время-затраты» называется *точкой полного уплотнения*.

> **ПРИМЕЧАНИЕ** Мой личный опыт показывает, что 30% с большой вероятностью является верхним пределом уплотнения по времени для любого программного проекта, причем даже этот уровень достигается с трудом. Вы можете воспользоваться этим порогом для проверки любых ограничений по срокам. Например, если проект имеет нормальное решение на 12 месяцев, а дедлайн установлен равным 7 месяцам, то этот проект построить не удастся, потому что он требует 41% уплотнения графика.

Дискретное моделирование

На реальной кривой «время-затраты», изображенной на рис. 9.3, между нормальным решением и решением минимальной продолжительности находит-

ся бесконечное количество точек. Конечно, никто не располагает временем для проектирования бесконечного количества решений, да и в этом нет необходимости. Вместо этого архитектор и менеджер проекта должны представить руководству одну-две точки между нормальным решением и решением минимальной продолжительности. Эти варианты представляют разумные сочетания времени и затрат, из которых руководство может выбрать наиболее подходящий вариант, и они всегда являются результатом некоторого уплотнения сети. В результате кривая, которая будет построена в ходе планирования проекта, является дискретной моделью (рис. 9.4). Хотя кривая «время-затраты» на рис. 9.4 содержит гораздо меньше точек, чем на рис. 9.3, она дает достаточно информации для того, чтобы правильно понять поведение проекта.

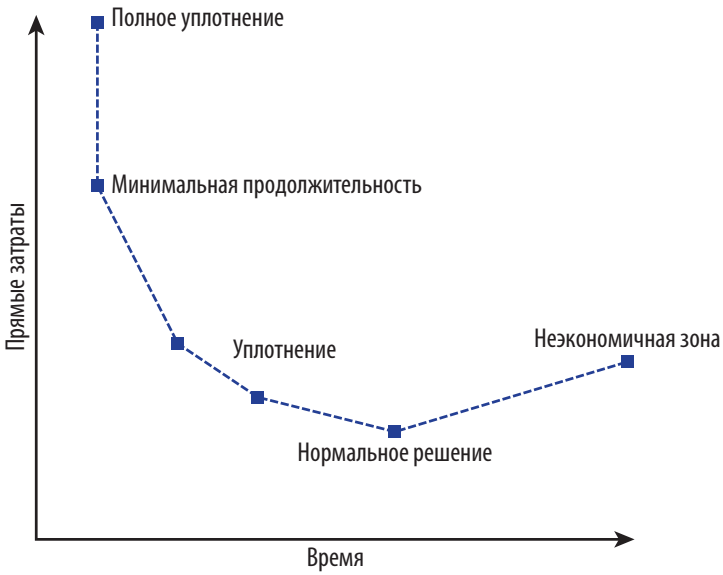


Рис. 9.4. Дискретная кривая «время-затраты» [адаптировано по материалам James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990)]

Предотвращение классических ошибок

Непрактичные решения с полным уплотнением и неэкономичные решения также стоит представить руководству, потому что многие руководители просто не знают об их непрактичности. У них вполне может быть ошибочная мысленная модель поведения проекта — скорее всего, сходная с изображенной на рис. 9.2. А с ошибочной мысленной моделью всегда принимаются ошибочные решения.

Допустим, график имеет наивысший приоритет, и руководитель готов на любые затраты ради соблюдения обязательств. Руководитель может думать, что выделение дополнительных средств и людей поможет команде продвинуться ближе к дедлайну, хотя никакие деньги не способны реализовать проект до минимальной продолжительности.

Также часто встречаются руководители с ограниченным бюджетом, но более свободным графиком. Такой руководитель может попытаться сократить затраты за счет субкритического комплектования проекта или непредоставления проекту необходимых ресурсов. При этом проект перемещается справа от нормального решения в неэкономичную зону, что приводит к значительному возрастанию затрат.

Реализуемость проекта

Кривая «время-затраты» отражает важнейший аспект проекта: его *реализуемость*. Планы проекта с сочетаниями времени и затрат, представляющими точки на кривой или над ней, могут быть воплощены в жизнь. Для примера возьмем точку A на рис. 9.5. Решение A требует времени T_2 и затрат C_1. Хотя решение A жизнеспособно, оно является субоптимальным. Если T_2 лежит в рамках допустимого дедлайна, то проект также может быть реализован с затратами C_2 — значение кривой «время-затраты» на момент времени T_2. Так как точка A лежит выше кривой, отсюда следует, что $C_2 < C_1$. И наоборот, если затраты C_1 приемлемы, для тех же затрат проект также может быть реализован за время T_1 — значение кривой «время-затраты» для времени C_1. Так как A находится справа от кривой, отсюда следует, что $T_1 < T_2$.

Точки на кривой «время-затраты» просто представляют оптимальное соотношение затрат и времени. Кривая «время-затраты» оптимальна, потому что всегда лучше реализовать проект быстрее (при тех же затратах) или с меньшими затратами (при том же сроке). Проект может быть реализован хуже, но не лучше кривой «время-затраты».

Также отсюда следует, что точки под кривой «время-затраты» невозможны. Например, возьмем точку B на рис. 9.6. Решение B требует времени T_3 и затрат C_4. Однако реализация проекта за время T_3 потребует затрат не менее C_3. Так как точка B находится под кривой «время-затраты», отсюда следует, что $C_3 > C_4$. Если вы не можете позволить себе затраты выше C_4, то проект потребует времени не менее T_4. Так как точка B находится слева от кривой «время-затраты», отсюда следует, что $T_4 > T_3$.

Мертвая зона

Если точки на кривой «время-затраты» представляют минимальные затраты для любой продолжительности, кривая «время-затраты» делит плоскость на две зоны. Первая — зона возможных решений — объединяет решения на кри-

вой «время-затраты» и выше нее. Вторая — мертвая зона — объединяет все решения ниже кривой «время-затраты» (рис. 9.7).

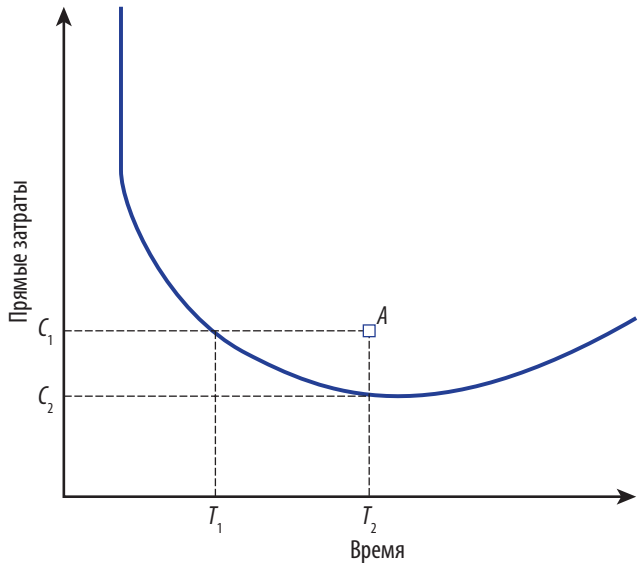


Рис. 9.5. Субоптимальное решение над кривой «время-затраты»

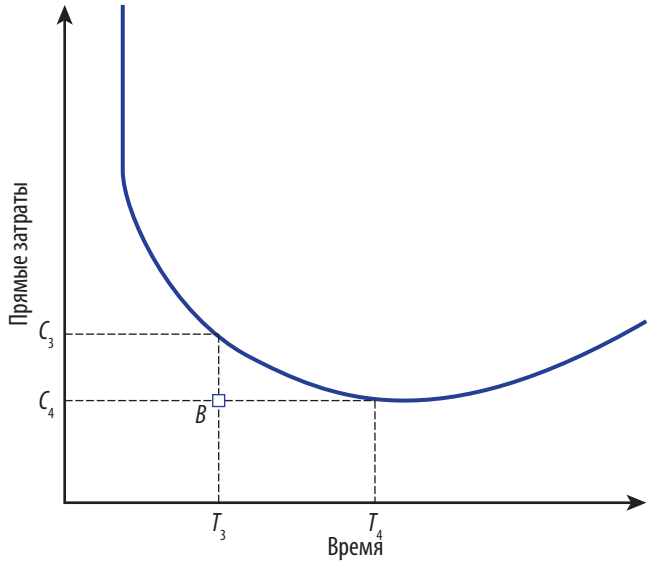


Рис. 9.6. Невозможное решение под кривой «время-затраты»

Рис. 9.7. Мертвая зона

Невозможно описать словами, насколько важно не спланировать проект в мертвой зоне. Проекты в мертвой зоне обречены на неудачу еще до того, как будет написана первая строка кода. Ключ к успеху — не архитектура и не технология, а предотвращение планирования проекта в мертвой зоне.

Поиск нормальных решений

При поиске нормального решения минимальный уровень комплектования, необходимый для беспрепятственного продвижения по критическому пути, часто неизвестен заранее. Например, тот факт, что проект может быть укомплектован двенадцатью разработчиками, не означает, что то же самое нельзя было сделать с восемью или даже шестью разработчиками без задержки проекта. Следовательно, поиск нормального уровня комплектования является итеративным процессом (рис. 9.8). При каждой итеративной попытке поиска нормального решения вы последовательно преобразуете все большую долю временного резерва в ресурсы. Такая замена естественным образом повышает риск проекта из-за сокращения временного резерва. Она также означает, что истинное нормальное решение уже содержит значительный уровень риска. Тем не менее наименьший уровень комплектования, необходимый для истинного нормального решения, часто достаточно хорош в отношении риска, пото-

му что для выполнения обязательств проекта еще остается достаточный объем временных резервов.

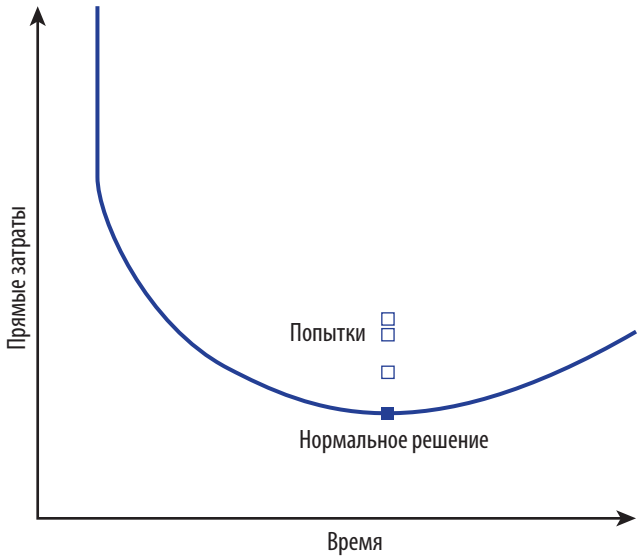


Рис. 9.8. Поиск нормального решения

Поправки на реальность

При определении уровня комплектования нормального решения следует делать небольшие поправки на реальность. Например, если в годичном проекте можно избежать найма еще одного работника, продлив график всего на неделю, вероятно, вам стоит пойти на такой компромисс. Также следует искать возможности упрощения выполнения проекта или сокращения риска интеграции в обмен на небольшое увеличение продолжительности или незначительное возрастание затрат. Всегда следует отдавать предпочтение таким поправкам на реальность. В результате промежуточные попытки могут не располагаться в точности друг над другом (как на рис. 9.8), а слегка смещаться вправо или влево.

Что является незначительной поправкой на реальность, а что должно рассматриваться как искажение исходных намерений при поиске нормального решения? Объективного ответа не существует. По моему эмпирическому правилу, все, что меньше 2–3% сроков или затрат, стоит интерпретировать как шум. Порог в 2–3% связан с дискретизацией планирования проекта и отслеживания прогресса. Если детализация активностей составляет неделю и если состояние проекта отслеживается на еженедельной основе, то за год

дискретизация планирования и измерений составит 2%, а все более точные данные должны интерпретироваться как шум. Эти соглашения продемонстрированы в главах 11 и 13.

Составляющие затрат проекта

До настоящего момента обсуждение затрат проекта было весьма упрощенным, потому что общие затраты по проекту складываются из двух составляющих: прямых затрат и непрямых затрат. При планировании проекта следует вычислить обе составляющие затрат вместе с общими затратами проекта. Понимание взаимодействия между составляющими затрат проекта исключительно важно для серьезного планирования проекта и принятия решений.

Прямые затраты

К прямым затратам по проекту относятся активности, которые добавляют напрямую измеряемую ценность в проект. Это те самые явные активности проекта, представленные на диаграмме планируемой осваиваемой ценности. Как объяснялось в главе 7, запланированная осваиваемая ценность (а следовательно, и прямые затраты) изменяется на протяжении жизненного цикла проекта, что приводит к пологой S-кривой.

Прямые затраты по программному проекту обычно включают следующие элементы:

- Разработчики, занимающиеся разработкой сервисов.
- Тестировщики, занимающиеся тестированием системы.
- Архитектор базы данных, проектирующий базу данных.
- Инженеры по тестированию, занимающиеся проектированием и построением тестовой оснастки.
- Эксперты UI/UX, проектирующие пользовательский интерфейс.
- Архитектор, занимающийся проектированием системы или планированием проекта.

Примерный вид кривой прямых затрат проекта показан на рис. 9.3.

Косвенные затраты

К косвенным затратам проекта относятся активности, которые добавляют в проект косвенную ценность, которую невозможно объективно измерить. Как

правило, такие активности присутствуют постоянно и не отображаются на диаграммах освоенной ценности или в плане проекта.

Косвенные затраты программного проекта обычно включают следующие элементы:

- Основная команда (то есть архитектор, менеджер проекта и менеджер продукта) после анализа SDP.
- Текущее управление конфигурацией, ежедневные сборки и ежедневное тестирование, или DevOps вообще.
- Отпуска и выходные.
- Закрепленные ресурсы между назначениями.

Косвенные затраты во многих проектах в целом пропорциональны продолжительности проекта. Чем больше времени занимает проект, тем выше косвенные затраты. Если построить график зависимости косвенных затрат от времени, он должен выглядеть как прямая линия.

Было бы неправильно думать о косвенных затратах как о ненужных потерях. Проект обречен на неудачу без архитектора и менеджера проекта, однако после анализа SDP они не имеют явно определенных активностей в плане.

Бухгалтерия и ценность

Концепции прямых и косвенных затрат перегружены смыслами. Одни рассматривают прямые затраты как затраты, связанные с непосредственными участниками команды, а косвенные — как затраты на внешних консультантов или субподрядчиков. Другие определяют прямые затраты как те, за которые должны платить они, а косвенные — как те, за которые должны платить другие люди или организации. Тем не менее вопрос о том, кто в конечном итоге должен платить за ресурсы, относится к бухгалтерии, а не к планированию проекта. Определения этой главы даются строго с точки зрения ценности: добавляет ли ресурс или активность измеряемую или неизмеряемую ценность?

Снова о мертвой зоне

Как и в случае с кривой прямых затрат, решения над кривой общих затрат реализуемы, а решения под ней невозможны. Таким образом, область под кривой общих затрат является настоящей мертвой зоной проекта, потому что она учитывает как прямые затраты, так и косвенные. Нахождение выше мертвой зоны на кривой прямых затрат еще не означает, что проекту ничего не угрожает, потому что вам приходится оплачивать косвенные затраты. Потратьте время на моделирование кривой общих затрат, а затем просто определите, оставляют ли

переданные вам параметры какие-либо шансы на успех. В главе 11 будет показано, как это делается.

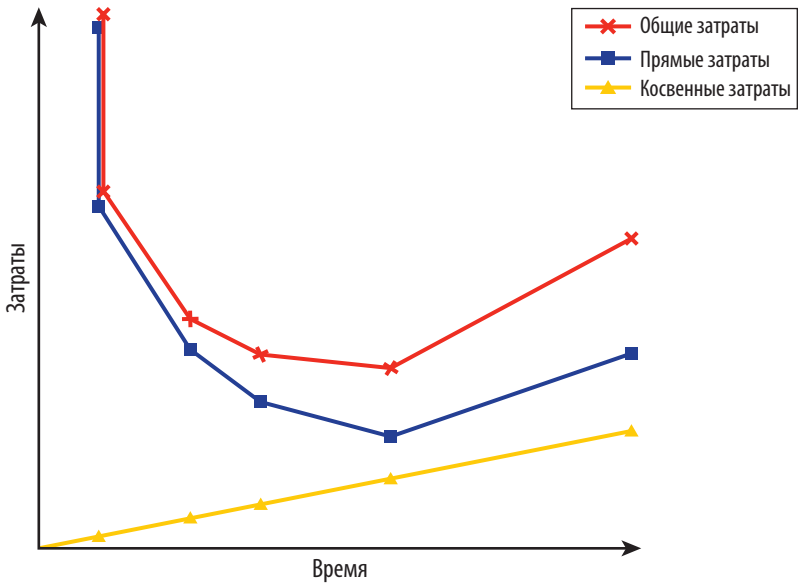


Рис. 9.9. Кривые прямых, косвенных и общих затрат проекта [адаптировано по материалам James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 4th ed. (Wiley, 1990)]

Уплотнение и составляющие затрат

План проекта с уплотнением сокращает продолжительность работы над проектом и, как следствие, также сокращает косвенные затраты на проект. В свою очередь, это обычно приводит к компенсации затрат на уплотнение проекта. Например, на рис. 9.9 обратите внимание на сегменты всех трех кривых между нормальным решением (самая нижняя точка кривой прямых затрат) и сжатым решением слева от нее. На кривой прямых затрат уплотненное решение имеет существенные дополнительные затраты между этими двумя точками. Однако на кривой общих затрат после учета косвенных затрат различия в общей стоимости заметно сокращаются. За небольшое возрастание общей стоимости между нормальным решением и первой точкой уплотнения вы получаете то же сокращение графика. Накопление косвенных затрат делает уплотнение более привлекательным по крайней мере изначально, потому что уплотнение обычно окупает себя. Во многих проектах сокращение косвенных затрат может оказаться еще более важным преимуществом, чем уплотнение графика.

ПЕРВАЯ КРИВАЯ «ВРЕМЯ-ЗАТРАТЫ» В ИСТОРИИ

Для совершенствования отрасли разработки в основном привлекаются инженерные идеи и практики, появившиеся в других отраслях. Кривая «время-затраты» является исключением из правила — она впервые появилась в области компьютеров и программирования.

В начале 1960-х годов компания General Electric разработала GE-225 — первый коммерческий компьютер на транзисторах.[1] Проект GE-255 ознаменовал технологический прорыв для своего времени. Именно в нем появилась первая операционная система с разделением времени (которая повлияла на архитектуру всех современных операционных систем), прямой доступ к памяти и язык программирования BASIC.

В 1960-е годы компания General Electric опубликовала статью с аналитическими выводами относительно связи между временем и затратами, сделанными на основании проекта GE-225.[2] В этой статье была приведена первая кривая «время-затраты» (аналогичная изображенной на рис. 9.4), а также представлено деление затрат на прямые и косвенные (см. рис. 9.9). Эти идеи быстро прижились в строительной отрасли.[3]

Джеймс Келли, соавтор первой статьи о методе критического пути, был консультантом в проекте GE-225. Между прочим, архитектором GE-225 был Арнольд Спилберг (Arnold Spielberg[4]), отец кинорежиссера Стивена Спилберга. В 2006 году общество IEEE наградило Спилберга престижной премией Основоположника компьютерных технологий.

Нормальное решение и минимальные общие затраты

На кривой прямых затрат нормальная точка по определению также является решением с минимальными затратами. Справа находится неэкономичная зона, а слева — уплотненные решения, которые добиваются выигрыша по времени ценой дополнительных затрат. Тем не менее после прибавления косвенных затрат для вычисления общих затрат по проекту для каждого из запланированных решений решение с минимальной общей стоимостью уже не будет нормальным решением. Добавление косвенных затрат сдвигает точку мини-

---

1 https://ru.wikipedia.org/wiki/GE-200

2 Børge M. Christensen, «GE 225 and CPM for Precise Project Planning» (*General Electric Company Computer Department*, December 1960).

3 James O'Brien, Scheduling Handbook (McGraw-Hill, 1969); and James M. Antill and Ronald W. Woodhead, Critical Path in Construction Practice, 2nd ed. (Wiley, 1970).

4 https://www.ge.com/reports/jurassic-hardware-steven-spielbergs-father-was-a-computing-pioneer/

мальных общих затрат влево от нормального решения. Более того, чем круче наклон линии косвенных затрат, тем более значительным будет смещение точки минимальных общих затрат влево.

Например, взгляните на рис. 9.10. На кривой прямых затрат нормальное решение очевидно является точкой с минимальными затратами. Однако на кривой общих затрат точкой с минимальными затратами является первое уплотненное решение слева от нормального. В этом случае уплотнение проекта привело к фактическому сокращению затрат. В результате точка минимальных общих затрат проекта становится оптимальным вариантом планирования проекта с точки зрения «время-затраты», потому что в ней проект завершается быстрее обычного и с меньшими общими затратами.

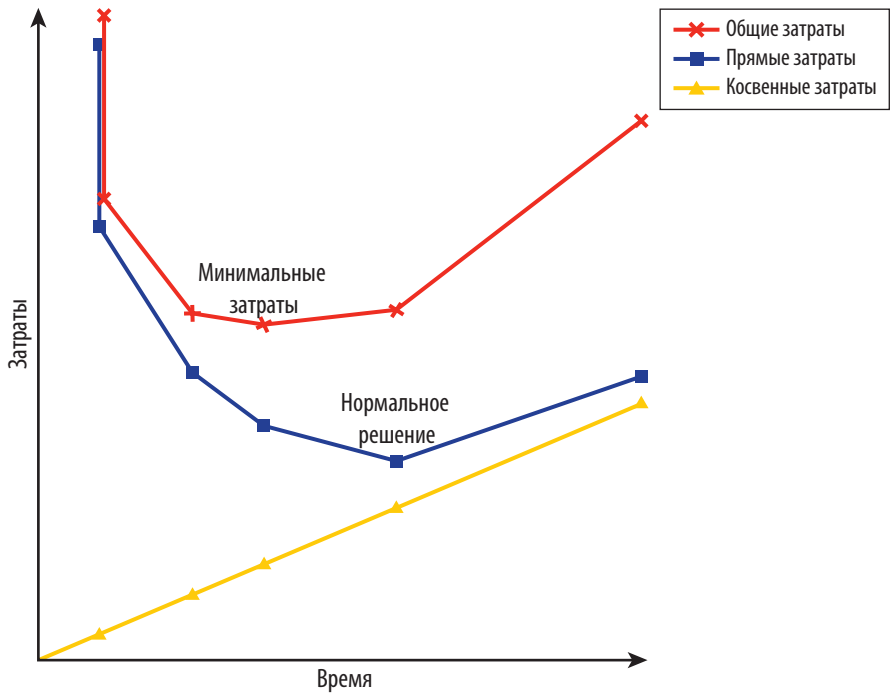


Рис. 9.10. Высокие косвенные затраты смещают минимальные общие затраты влево от нормального решения

На рис. 9.10 сдвиг влево от точки с минимальными общими затратами хорошо различим из-за дискретной природы диаграммы. Следует помнить, что сдвиг влево присутствует всегда, даже с непрерывными диаграммами вроде изображенной на рис. 9.11, у которой линия косвенных затрат получается более пологой, чем на рис. 9.10.

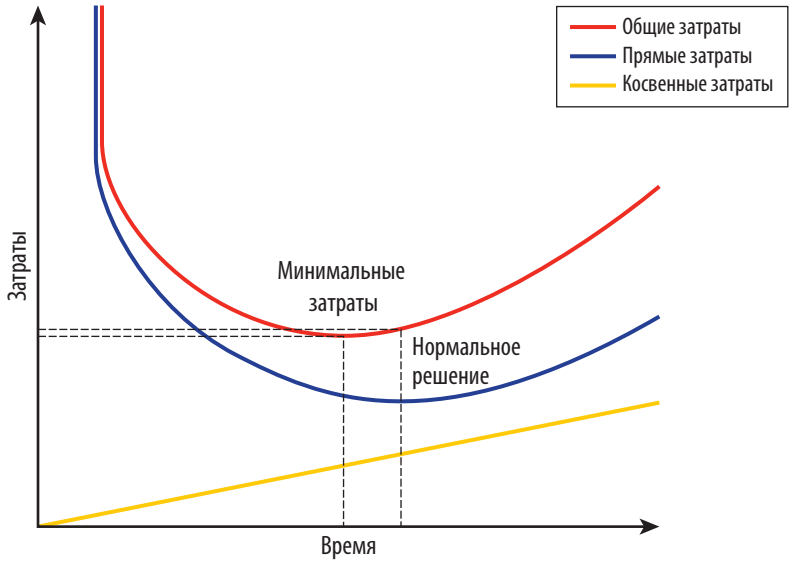


Рис. 9.11. Сдвиг влево на непрерывной кривой «время-затраты»

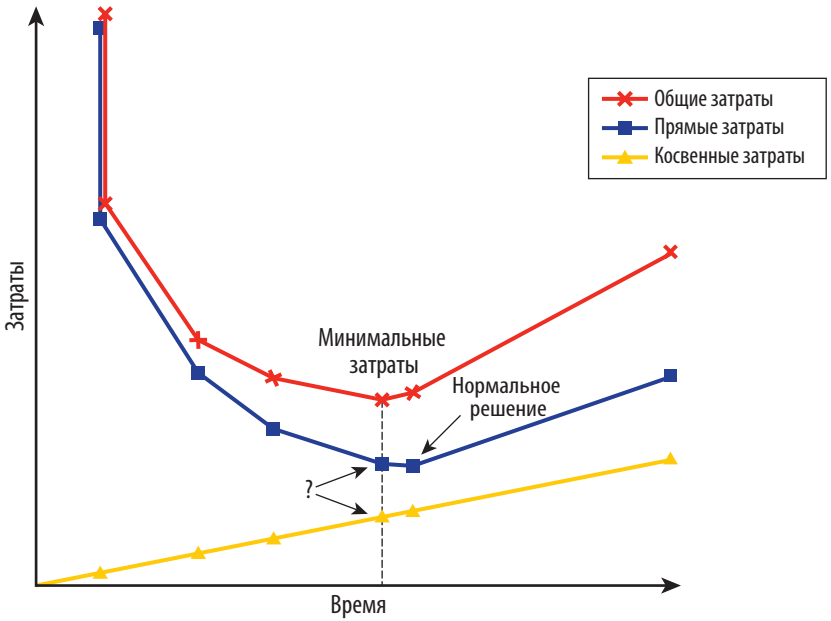


Рис. 9.12. Минимальные общие затраты как неизвестная точка

Так как вы будете разрабатывать лишь небольшое подмножество вариантов планирования проекта, построенная вами кривая «время-затраты» всегда будет дискретной моделью проекта. С имеющимися решениями (и уровнем косвенных затрат) нормальное решение действительно может быть точкой с минимальными общими затратами, как на рис. 9.9. Но такая ситуация обманчива: это всего лишь артефакт от упущенного решения, находящегося чуть левее нормального.

Рисунок 9.12 не отличается от рис. 9.9, за исключением того, что на нем добавлена неизвестная точка слева от нормального решения, которая демонстрирует сдвиг влево от точки с минимальными общими затратами.

В такой ситуации проблема в том, что вы не имеете ни малейшего представления о том, как прийти к этому решению: вы не знаете, какая комбинация ресурсов и уплотнения дает нужную точку. Хотя такое решение всегда существует в теории, на практике для большинства проектов общие затраты нормального решения можно приравнять к минимальным общим затратам проекта. Разность между общими затратами нормального решения и точкой истинного минимума общих затрат часто не оправдывает усилий, необходимых для поиска этой конкретной точки.

Косвенные затраты и риск

Чем круче угол наклона линии косвенных затрат, тем более значительным будет сдвиг влево от точки минимальных общих затрат. С высокими косвенными затратами одно из уплотненных решений с большой вероятностью станет оптимальной точкой проекта, потому что его затраты будут ниже, чем у нормального решения, при этом проект будет реализован в более короткий срок. Тем не менее возникает проблема: уплотненные решения по планированию проекта обычно сопряжены с более высоким риском. Риск может быть обусловлен как критичностью проекта, так и ростом сложности его выполнения. Следовательно, высокие косвенные затраты означают, что оптимальная точка проекта с большей вероятностью будет рискованным вариантом, что вряд ли можно считать рецептом успеха. Как следствие, проекты с высокими косвенными затратами почти всегда также являются рискованными. О том, как справиться с этими рисками, будет рассказано в следующих главах.

Персонал и составляющие затрат

В каждом из ваших вариантов планирования проектов должны учитываться как прямые, так и косвенные затраты. Как было установлено в главе 7, общие затраты программного проекта определяются площадью под диаграммой распределения комплектования. Если вы знаете общие затраты по проекту и одну из составляющих затрат (например, прямые затраты), другая составляющая затрат определяется вычитанием ее из общих затрат. Для каждого варианта планирования проекта вы сначала комплектуете проект, затем рисуете диаграмму

планируемой освоенной ценности и диаграмму планируемого распределения комплектования. Затем вы вычисляете размер области под диаграммой распределения комплектования для получения общих затрат, а также суммируете трудозатраты по всем активностям прямых затрат (тех, которые отображаются на диаграмме освоенной ценности). Косвенные затраты просто вычисляются как разность двух величин.

В графическом виде на рис. 9.13 изображена типичная структура составляющих затрат под диаграммой распределения комплектования (также см. рис. 7.8). В начальной стадии проекта задействована только основная команда, и большая часть этой работы включает косвенные затраты. Остальная работа, выполняемая основной командой, включает прямые затраты — например, проектирование системы и проекта. Тем не менее после анализа SDP основная команда переходит в категорию чистых косвенных затрат. После анализа SDP проект содержит постоянные косвенные затраты — DevOps, ежедневные сборки и ежедневное тестирование. Остальное комплектование относится к прямым затратам (например, разработчики, занимающиеся построением системы).

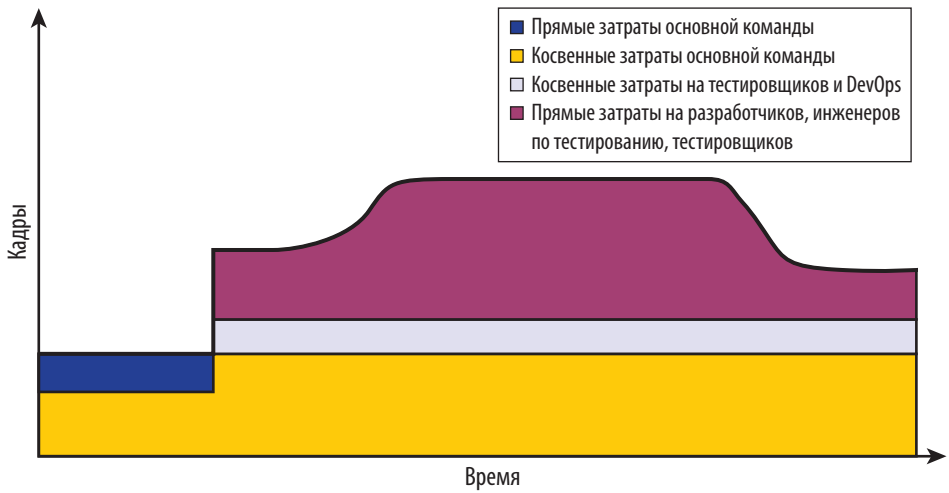


Рис. 9.13. Составляющие затрат под графиком распределения комплектования

Затраты прямые и косвенные

Как видно из рис. 9.13, в типичном программном проекте доля косвенных затрат выше, чем доля прямых затрат. Многие люди не осознают, сколько косвенных затрат требуется для разработки высококачественной сложной программной системы. Отношение прямых затрат к косвенным 1:2 вполне типично, но это отношение вполне может быть более высоким. Точное отношение прямых затрат к косвенным часто определяется природой бизнеса. Например, в компа-

нии, которая производит авиационные приборы, можно ожидать более высоких косвенных затрат по сравнению с той, которая производит обычные коммерческие системы.

Косвенные затраты и общие затраты

В программном проекте косвенные затраты часто преобладают в общих затратах. Это приводит к ключевому наблюдению: при прочих равных обстоятельствах более короткие проекты всегда обходятся дешевле, потому что они создают меньшее количество косвенных затрат. Это утверждение истинно независимо от того, как достигается уплотнение графика — уплотнением проекта или применением практик, перечисленных в начале этой главы. Более короткий проект обходится дешевле даже в том случае, если уплотнение требует дополнительных и даже более дорогих ресурсов.

К сожалению, многие руководители просто не знают, что более короткие проекты обходятся дешевле, что приводит к классической ошибке. Сталкиваясь с ограниченным бюджетом, руководитель пытается сократить затраты за счет регулировки ресурсов (то есть снижения их качества или количества). Это затягивает работу над проектом, так что в конечном итоге проект обходится намного дороже.

Фиксированные затраты

У программных проектов есть еще одна составляющая затрат, фиксированная по времени. Фиксированные затраты могут включать компьютерное оборудование или лицензии на программные продукты. Фиксированные затраты проекта выражаются постоянным подъемом прямой косвенных затрат (рис. 9.14).

Так как фиксированные затраты просто сдвигают кривую «время-затраты» вверх, они ничего не добавляют к процессу принятия решений, так как влияют на все варианты практически одинаково (хотя влияние может слегка зависеть от размеров команды). В большинстве программных проектов сколько-нибудь серьезного размера фиксированные затраты составят приблизительно 1–2% от общих затрат, так что их можно считать пренебрежимо малыми.

Уплотнение сети

Уплотнение проекта изменяет его сеть. Уплотнение должно быть итеративным процессом, в котором вы постоянно ищете лучший следующий шаг. Все начинается с нормального решения. Нормальное решение должно хорошо реагировать на уплотнение, потому что оно находится на минимуме кривой «время-затраты». Как упоминалось ранее, на первых шагах уплотнение даже

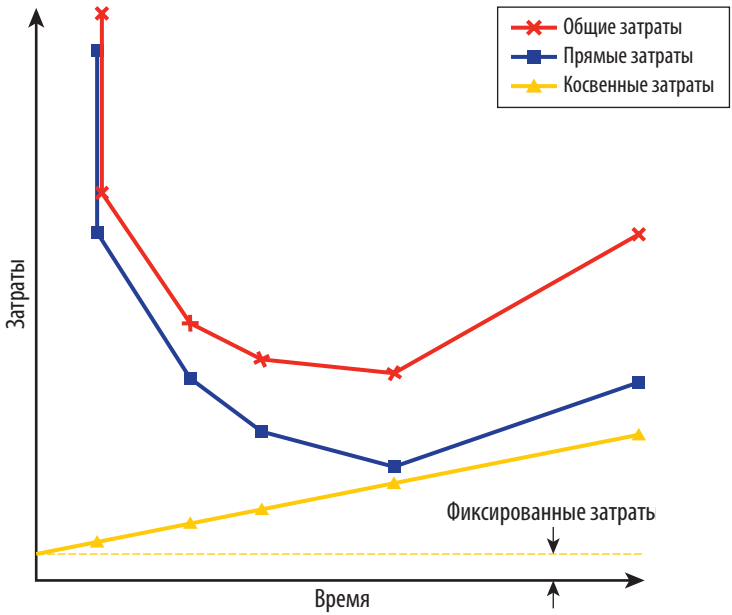


Рис. 9.14. Добавление фиксированных затрат

может окупить само себя. Кроме того, непосредственно слева от нормального решения кривая «время-затраты» оказывается самой плоской. Таким образом, первые одна-две точки уплотнения обеспечат наилучшую окупаемость затрат на уплотнение. Тем не менее при дальнейшем уплотнении проекта вы начнете взбираться вверх по кривой «время-затраты» и в конечном счете столкнетесь с тем, что затраты на уплотнение перестают окупаться. В проекте будет все меньше уплотнения графика при непрерывно растущих затратах, так, будто он начинает сопротивляться дальнейшему уплотнению. При уплотнении проекта в целом вы должны попытаться умножить эффект, уплотняя ранее уплотненное решение, а не просто применяя новый метод сжатия к базовому нормальному решению.

Поток уплотнения

Следует по возможности избегать попыток уплотнения активностей, которые плохо реагируют на уплотнение независимо от затрат (таких, как архитектура) или уже уплотненных активностей. Так как даже отдельные активности имеют собственную кривую «время-затраты», поначалу активность может уплотняться легко, но последующее сжатие потребует дополнительных затрат для восхождения по собственной кривой «время-затраты» активности. В какой-то момент дальнейшее уплотнение активности станет невозможным. По этой

причине в общем случае лучше уплотнять другие активности, чем повторно уплотнять одну и ту же активность.

В идеале следует уплотнять только активности, лежащие на критическом пути. Вряд ли есть смысл сжатия активностей за пределами критического пути, потому что это приведет к росту затрат без уменьшения сроков. В то же время не нужно бездумно уплотнять все активности на критическом пути. Лучшими кандидатами на уплотнение являются активности, обеспечивающие лучшую результативность для уплотнения. Уплотнение этих активностей обеспечивает наибольшее сокращение сроков для минимальных дополнительных затрат. Продолжительность активности также важна, потому что все методы уплотнения повышают риск и сложность проекта. Лучше возложить эти эффекты на большую критическую активность и получить наибольшее сокращение сроков. Также в общем случае рекомендуется расщепить большие активности на меньшие — приятный побочный эффект от уплотнения большой активности.

Уплотнение критического пути приводит к его сокращению. В результате самым длинным в сети проекта может стать другой путь; иначе говоря, появляется новый критический путь. Постоянно оценивайте сеть проекта, чтобы обнаружить факт появления нового критического пути и уплотнить этот путь вместо старого критического пути. Если появится сразу несколько критических путей, вы должны искать возможности их уплотнения параллельно и на равные величины. Например, если активность или набор активностей покрывают все критические пути, то именно на них будет направлена следующая итерация уплотнения.

Уплотнение проекта может повторяться многократно до выполнения одного из следующих условий:

- Достигнут желательный дедлайн, поэтому нет смысла планировать еще более дорогостоящие и короткие проекты.
- Вычисляемые затраты проекта превышают бюджет, выделенный для проекта.
- Уплотненная сеть проекта становится настолько сложной, что проект вряд ли может быть реализован каким-либо менеджером проекта или командой.
- Продолжительность уплотненного решения более чем на 30% (или даже на 25%) меньше продолжительности нормального решения. Как упоминалось ранее, на практике существует естественный предел для уплотнения любого проекта.
- Уплотненные решения слишком рискованны или риск постепенно снижается, потому что точка максимального риска уже пройдена. Это снижает возможность численной оценки риска планов проекта (см. следующую главу).

- У вас кончились идеи или варианты дальнейшего уплотнения проекта, то есть уплотнять больше нечего.
- Появилось слишком много критических путей, или все сетевые пути стали критическими.
- Возможности уплотнения активностей обнаружены только за пределами критического пути. Уплотненное решение имеет такую же продолжительность, как и предыдущее, но оно стоит дороже. Иначе говоря, достигнута точка полного уплотнения проекта.

Понимание проекта

Серия уплотненных решений позволяет лучше смоделировать проект и понять, как он ведет себя в условиях изменений его граничных условий по времени и затратам. Часто бывает достаточно всего двух или трех точек слева от нормального решения, чтобы понять, как ведет себя проект. Чем сложнее или затратнее проект, тем больше следует потратить на его понимание, потому что даже ничтожные ошибки имеют очень серьезные последствия.

10 Риск

Как было показано в главе 9, каждый проект всегда имеет несколько вариантов планирования с разными сочетаниями времени и затрат. Некоторые из этих вариантов с большой вероятностью окажутся более агрессивными или рискованными, чем другие. По сути, каждый вариант плана проекта соответствует некоторой точке трехмерного пространства, координатными осями которого являются время, затраты и риск. Ответственные за принятие решений должны уметь учитывать риски при выборе варианта планирования проекта. Таким образом, при планировании проекта вы должны уметь выражать риски разных вариантов в числовой форме.

Многие люди понимают, что ось рисков существует, но игнорируют ее, потому что не могут измерить ее значения или выразить в числовой форме. Это неизбежно приводит к плохим результатам, обусловленным применением двумерной модели (время и затраты) к трехмерной задаче (время, затраты и риск). В этой главе объясняется, как просто и объективно измерить риски при помощи некоторых методов моделирования. Вы узнаете, как риск соотносится с временем и затратами, как сократить риск проекта и как найти оптимальную точку планирования для проекта.

> **ПРИМЕЧАНИЕ** Вычисление рисков основано на простой математике. Чтобы автоматизировать алгебраическую основу и избежать ручных вычислений с повышенным риском ошибок, в прилагаемых файлах по адресу rightingsoftware.org содержатся примеры в форме электронных таблиц для вычисления риска.

Выбор вариантов

Конечная цель моделирования рисков — оценка вариантов планирования проекта в свете рисков, а также времени и затрат с целью определения реализуемости этих вариантов. В общем случае риск является лучшим критерием для выбора между вариантами.

ТЕОРИЯ ПЕРСПЕКТИВ

В 1979 году психологи Дэниел Канеман и Амос Тверски разработали теорию перспектив[1] — одну из важнейших концепций поведенческой психологии в области принятия решений. Канеман и Тверски обнаружили, что люди принимают решения на основании возможного риска, а не ожидаемых приобретений. При измеряемых идентичных потерях или приобретениях большинство людей страдают от потерь непропорционально сильнее, чем радуются от тех же приобретений. В результате люди стремятся сократить риски, а не максимизировать приобретения, даже если было бы логичнее пойти на риск. Эти наблюдения противоречат распространенному мнению, что люди действуют рационально с целью максимизации приобретений на основании ожидаемой ценности. Теория перспектив подчеркивает важность добавления риска к времени и затратам в процессе принятия решений. В 2002 году Дэниел Канеман получил Нобелевскую премию по экономике за свою работу, развивающую теорию перспектив.

Для примера рассмотрим два варианта одного проекта: первый требует 12 месяцев и 6 разработчиков, а второй — 18 месяцев и 4 разработчиков. Если это все, что вам известно о двух вариантах, большинство людей выберет первый вариант, потому что оба варианта в итоге имеют одинаковые затраты (6 человеколет), а первый вариант реализуется намного быстрее (при условии, что вы располагаете необходимым текущим финансированием). Теперь предположим, что вы знаете, что первый имеет всего 15-процентную вероятность успеха, а второй — 70-процентную вероятность. Какой вариант вы выберете на этот раз? Возьмем более яркий пример: допустим, второй вариант требует 24 месяцев и 6 разработчиков при тех же 70% успеха. Хотя второй вариант теперь стоит вдвое дороже и занимает больше времени, большинство людей интуитивно выберет этот вариант. Это простая демонстрация того, что при выборе варианта люди часто руководствуются риском, а не временем и затратами.

Кривая «время-риск»

Наряду с кривой «время-затраты» у проектов также имеется кривая «время-риск». Идеальная кривая обозначена на рис. 10.1 пунктирной линией.

При уплотнении проекта более короткие варианты планирования сопряжены с повышенным уровнем риска, и скорость роста с большой вероятностью нелинейна. Вот почему пунктирная линия на рис. 10.1 устремляется вверх при приближении к вертикальной оси риска и медленно убывает с ростом времени. Тем не менее интуитивная пунктирная линия неверна. На практике кри-

[1] Daniel Kahneman and Amos Tversky, “Prospect Theory: An Analysis of Decision under Risk,” Econometrica, 47, no. 2 (March 1979): 263–292.

вая «время-затраты» представляет собой разновидность логистической функции — сплошная линия на рис. 10.1.

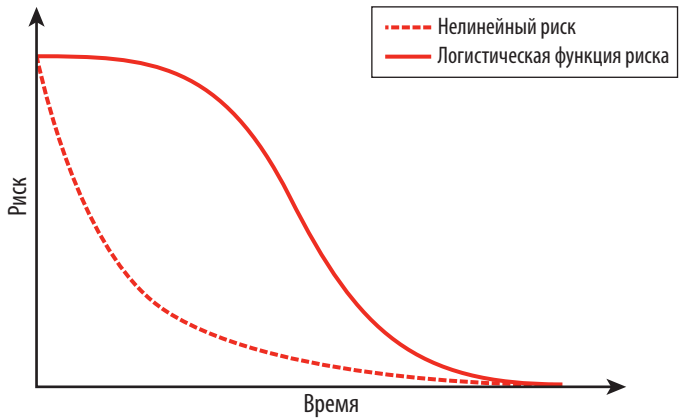


Рис. 10.1. Идеальная кривая «время-риск»

Логистическая функция является более совершенной моделью, потому что она лучше отражает общее поведение риска в сложных системах. Например, если бы я захотел построить график риска того, что мой ужин подгорит из-за попыток уплотнить нормальное время приготовления, то кривая риска напоминала бы сплошную линию на рис. 10.1. Каждый метод уплотнения — завышение температуры в духовке, размещение противня слишком близко к нагревательному элементу, выбор более простой в приготовлении, но быстрее подгорающей еды, пропуск предварительного нагрева духовки и т. д. — повышает риск того, что обед подгорит. Как показывает сплошная линия, риск подгорания ужина из-за накапливаемого уплотнения в какой-то момент почти максимизируется и даже выравнивается, потому что ужин почти наверняка подгорит. И наоборот, если я решу вообще не входить на кухню, риск резко сократится. Если бы риск описывался пунктирной линией, то у меня всегда бы оставался шанс избежать подгорания ужина, так как я всегда бы мог повысить риск за счет его дальнейшего уплотнения.

Учтите, что логистическая функция всегда имеет точку перегиба, в которой риск радикально возрастает (аналог решения войти на кухню). Напротив, пунктирная линия продолжает постепенно расти и не имеет заметной точки перегиба.

Реальная кривая «время-риск»

Даже логистическая функция на рис. 10.1 все еще остается идеализированной кривой «время-риск». Реальная кривая «время-риск» более напоминает изображенную на рис. 10.2. Причина такой формы кривой лучше всего объясня-

ется наложением ее на кривую прямых затрат проекта. Так как поведение проекта имеет трехмерную природу, на рис. 10.2 используется вторичная ось *y* для риска.

Рис. 10.2. Реальная кривая «время-затраты-риск»

Вертикальная пунктирная линия на рис. 10.2 обозначает продолжительность нормального решения, а также минимальное решение прямых затрат для проекта. Учтите, что нормальное решение обычно обменивает некоторое количество временных резервов для снижения комплектования. Сокращение временного резерва проявляется в повышении уровня риска.

Слева от нормального решения находятся более короткие, уплотненные решения. Уплотненные решения также являются более рискованными, так что кривая риска возрастает слева от нормального решения. Риск сначала растет, а затем выравнивается (как в случае с идеальной логистической функцией). Тем не менее, в отличие от идеального поведения, реальная кривая риска максимизируется перед точкой минимальной продолжительности и даже немного убывает, приобретая вогнутую форму. Хотя такое поведение является нелогичным, оно объясняется тем, что в общем случае более короткие проекты являются более безопасными — феномен, который я называю *эффектом да Винчи*. При изучении сопротивления растяжению проволоки Леонардо да Винчи обнаружил, что короткие провода прочнее длинных (возможно, потому, что вероятность дефекта пропорциональна длине провода[1]). То же самое мож-

[1] William B. Parsons, *Engineers and Engineering in the Renaissance* (Cambridge, MA: MIT Press, 1939); Jay R. Lund and Joseph P. Byrne, Leonardo da Vinci's Tensile Strength Tests: Implications for the Discovery of Engineering Mechanics (Department of Civil and Environmental Engineering, University of California, Davis, July 2000).

но сказать и о проектах. Чтобы продемонстрировать этот момент, рассмотрим два возможных способа реализации проекта на 10 человеко-лет: 1 человек на 10 лет или 3650 людей на 1 день. Если предположить, что оба варианта жизнеспособны (что люди доступны, что вы располагаете необходимым временем и т. д.), однодневный проект намного безопаснее десятилетнего. Вероятность того, что что-то плохое произойдет за один день, вычислить довольно трудно, но за 10 лет это произойдет почти наверняка. Позднее в этой главе я приведу более объективное объяснение такого поведения.

Справа от нормального решения риск убывает, по крайней мере изначально. Например, выделение лишней недели на проект, рассчитанный на один год, сократит риск несоблюдения срока. Но если выделить на проект больше времени, в какой-то момент вступит в силу закон Паркинсона, а риск резко повысится. Таким образом, справа от нормального решения кривая риска убывает, минимизируется при некотором значении, которое будет больше нуля, а затем снова начинает возрастать, в результате чего график принимает вогнутую форму.

Моделирование риска

В этой главе представлены мои методы моделирования и числового выражения риска. Эти модели дополняют друг друга в отношении того, как они измеряют риск. Часто для выбора между вариантами требуется более одной модели — ни одна модель никогда не будет идеальной. Тем не менее все модели рисков дают сопоставимые результаты.

Значения рисков всегда относительны. Например, прыжок с быстро движущегося поезда рискован. С другой стороны, если поезд мчится к краю пропасти, спрыгнуть с него — самое разумное, что только можно сделать. Риск не имеет абсолютного значения, поэтому его можно оценить только в сравнении с другими альтернативами.

Таким образом, есть смысл говорить о «более рискованном» проекте, а не о «рискованном». Также ничто не бывает полностью безопасным. Единственный абсолютно безопасный подход к любому проекту — не браться за него. Таким образом, можно говорить о «более безопасном» проекте, а не просто о «безопасном».

Нормализация риска

Вся суть оценки рисков — возможность сравнения вариантов и проектов, основанная на сравнении чисел. Первое решение, которое я принял при создании моделей, — нормализация рисков в числовом диапазоне от 0 до 1.

Значение риска 0 не означает, что проект полностью лишен риска. Оно означает лишь то, что риск проекта был минимизирован. Аналогичным образом зна-

чение 1 не означает, что проект обречен на неудачу, а лишь то, что риск проекта достиг максимума.

Значение риска также не определяет вероятность успеха. В теории вероятностей значение 1 означает полную определенность, а значение 0 — невозможность. Проект с риском 1 может реализоваться, а проект с риском 0 еще может завершиться неудачей.

Риски и временные резервы

Временные резервы различных активностей в сети предоставляют объективные средства измерения рисков по проекту, а в предыдущих главах временные резервы упоминались при обсуждении риска. Два разных варианта проекта различаются по временным резервам, а следовательно, могут радикально различаться по рискам. Для примера возьмем два параметра проекта на рис. 10.3.

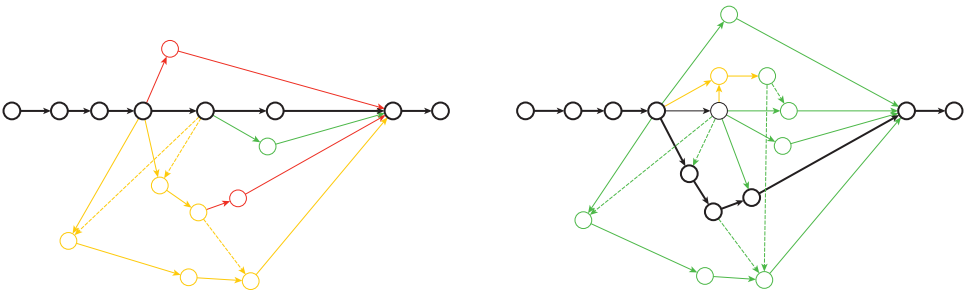


Рис. 10.3. Два варианта проекта

Оба варианта являются действительными вариантами планирования проекта для построения одной системы. Единственная информация, доступная на рис. 10.3, — временные резервы двух сетей, закодированные в цветовой форме. Теперь спросите себя: в каком из двух проектов вы бы предпочли участвовать? Все, кому я показывал две диаграммы, выбирали на рис. 10.3 правый вариант. Интересно, что никто никогда не спрашивал о различиях между продолжительностью и затратами этих двух вариантов. Даже когда я говорил, что правый вариант занимает на 30% больше времени и обходится дороже, эта информация не влияла на выбор. Никто не выбирал проект с низкими временными резервами, высоким напряжением и высоким риском, показанным в левой части рис. 10.3.

Проектные риски

Ваш проект сталкивается с несколькими видами рисков. Существует риск комплектования (получит ли проект необходимый ему уровень кадров?).

Существует риск продолжительности (будет ли проекту выделен необходимый период времени?). Существует технологический риск (позволит ли технология реализовать решение?). Существует человеческий фактор (обладает ли команда достаточной технической компетенцией и смогут ли ее участники работать вместе?). Существует ли риск исполнения (сможет ли менеджер проекта правильно выполнить план проекта?).

Эти типы риска не зависят от разновидности риска, оцениваемого при помощи временных резервов. Любой план проекта всегда предполагает, что организация или команда будет располагать всем необходимым для реализации проекта с запланированными сроками и затратами и что проект получит требуемое время и ресурсы. Остальные риски относятся к тому, насколько хорошо проект справится с непредвиденным. Я называю такие риски *проектными рисками*.

Проектные риски описывают чувствительность проекта к отставанию от графика и вашему умению выполнять свои обязательства. Таким образом, проектные риски описывают непрочность проекта, то есть степень, в которой проект напоминает карточный домик. Использование временных резервов для измерения риска в действительности означает числовую оценку этого проектного риска.

Риск и прямые затраты

Измерения проектного риска обычно относятся к прямым затратам и продолжительности различных решений. В большинстве проектов косвенные затраты не зависят от риска по проекту. Косвенные затраты продолжают расти с продолжительностью проекта даже при очень низком риске. По этой причине в данной главе упоминаются только прямые риски.

Риск возникновения критичности

Модель риска возникновения критичности пытается выразить в числовой форме интуитивное впечатление о рисках при оценке вариантов на рис. 10.3. Для этой модели риска активности проекта классифицируются на четыре категории по убыванию риска:

- *Критические активности*. Очевидно, что критические активности являются самыми рискованными, потому что любая задержка в критической активности всегда приводит к нарушению ограничений по срокам и затратам.
- *Активности с высоким риском*. Околокритические активности с низким временным резервом также рискованны, потому что любая задержка в них также приводит к нарушению ограничений по срокам и затратам.
- *Активности со средним риском*. Активности со средним временным резервом обладают средним уровнем риска и могут столкнуться с задержками.

- *Активности с низким риском.* Активности с высоким временным резервом сопряжены с наименьшим риском и могут выдержать даже большие задержки без нарушения графика проекта.

Из анализа следует исключить активности нулевой продолжительности (например, контрольные точки и фиктивные активности), потому что они ничего не добавляют к риску проекта. Более того, в отличие от реальных активностей, они просто являются артефактами сети проекта.

Глава 8 показывает, как использовать цветовое кодирование для классификации активностей в зависимости от их временного резерва. Тот же метод может использоваться для оценки чувствительности или непрочности активностей по цвету с кодированием четырех категорий риска. Когда цветовое кодирование будет назначено, назначьте вес для критичности каждой активности. Вес играет роль фактора риска. Конечно, вы можете выбирать любые веса, описывающие различия в степени риска. Одно из возможных распределений весов показано в табл. 10.1.

Таблица 10.1. Веса рисков возникновения критичности

Цвет активности	Вес
Черный (критическая)	4
Красный (высокий риск)	3
Желтый (средний риск)	2
Зеленый (низкий риск)	1

Формула риска возникновения критичности выглядит так:

$$\text{Риск} = \frac{W_C \times N_C + W_R \times N_R + W_Y \times N_Y + W_G \times N_G}{W_C \times N},$$

где:

- W_C — вес черных активностей (критических);
- W_R — вес красных активностей (низкий временной резерв);
- W_Y — вес желтых активностей (средний временной резерв);
- W_G — вес зеленых активностей (высокий временной резерв);
- N_C — количество черных активностей (критических);
- N_R — количество красных активностей (низкий временной резерв);
- N_Y — количество желтых активностей (средний временной резерв);

- N_G — количество зеленых активностей (высокий временной резерв);
- N — количество активностей в проекте ($N = N_C + N_R + N_Y + N_G$).

После подстановки весов из табл. 10.1 формула риска возникновения критичности принимает следующий вид:

$$\text{Риск} = \frac{4 \times N_C + 3 \times N_R + 2 \times N_Y + 1 \times N_G}{4 \times N}.$$

Применение формулы риска возникновения критичности к сети на рис. 10.4 дает следующий результат:

$$\text{Риск} = \frac{4 \times 6 + 3 \times 4 + 2 \times 2 + 1 \times 4}{4 \times 16} = 0{,}69.$$

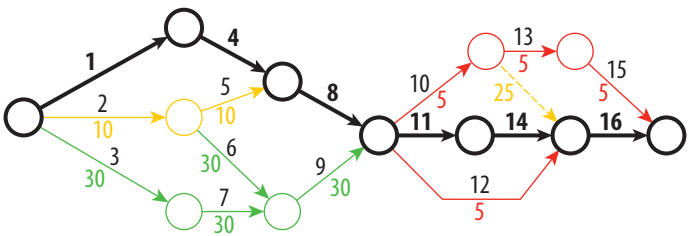


Рис. 10.4. Пример сети для вычисления риска

Значения риска возникновения критичности

Максимальное значение риска возникновения критичности равно 1,0; оно встречается в том случае, когда все активности в сети являются критичными. В такой сети N_R, N_Y и N_G равны нулю, а N_C равно N:

$$\text{Риск} = \frac{W_C \times N + W_R \times 0 + W_Y \times 0 + W_G \times 0}{W_C \times N} = \frac{W_C}{W_C} = 1{,}0\,.$$

Минимальное значение риска возникновения критичности равно отношению W_G к W_C; оно встречается в том случае, когда все активности в сети являются зелеными. В такой сети N_C, N_R и N_Y равны нулю, а N_G равно N:

$$\text{Риск} = \frac{W_C \times 0 + W_R \times 0 + W_Y \times 0 + W_G \times N}{W_C \times N} = \frac{W_G}{W_C}.$$

С весами из табл. 10.1 минимальное значение риска равно 0,25. Таким образом, риск возникновения критичности никогда не равен нулю: у таких взвешенных

средних минимальное значение больше нуля при условии, что сами веса больше нуля. Это не обязательно плохо, так как риск по проекту никогда не должен быть нулевым — и это логично, поскольку любой проект, которым стоит заниматься, сопряжен с риском.

Выбор весов

Если вы можете дать разумное обоснование для своего выбора весов, скорее всего, модель риска возникновения критичности сработает. Например, набор весов [21, 22, 23, 24] вряд ли можно назвать удачным, потому что 21 всего на 14% меньше 24; таким образом, этот набор не подчеркивает риск зеленых активностей по сравнению с критическими. Более того, минимальный риск с такими весами (W_G/W_C) равен 0,88, что явно слишком высоко. Я считаю, что набор [1, 2, 3, 4] ничем не хуже любого другого разумного выбора.

Настройка риска возникновения критичности

Модель риска возникновения критичности часто требует некоторой настройки и субъективных решений. Во-первых, как упоминалось в главе 8, диапазоны разных цветов (критерии красных, желтых и зеленых активностей) должны соответствовать продолжительности вашего проекта. Во-вторых, следует подумать об определении активностей с очень низким временным резервом или околокритических активностей (например, активностей с 1 днем временного резерва) как критических, потому что они имеют практически такой же риск, как и критические. В-третьих, даже если временные резервы некоторых активностей не являются околокритическими, вы должны проанализировать цепочку, в которой находятся эти активности, и отрегулировать ее соответствующим образом. Например, если у вас есть годовая цепочка из многих активностей, которая имеет всего 10 дней временного резерва, каждая активность в такой цепочке должна быть помечена как критическая для вычисления риска. Сбой одной активности в этой цепочке поглотит весь временной резерв, в результате чего все последующие активности станут критическими.

Риск Фибоначчи

В числовой последовательности Фибоначчи каждое число равно сумме двух предыдущих (кроме первых двух значений, которые по определению равны 1).

$$\mathrm{Fib}_n = \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2}$$
$$\mathrm{Fib}_2 = \mathrm{Fib}_1 = 1$$

Это рекурсивное определение дает последовательность 1, 1, 2, 3, 5, 8, 13, ...

Отношение двух (достаточно больших) последовательных чисел Фибоначчи стремится к иррациональному числу «фи» (греческая буква φ), значение которого равно 1,618..., и последовательность может быть выражена в следующем виде:

$$\text{Fib}_i = \varphi \times \text{Fib}_{i-1}.$$

С древних времен значение φ было известно под названием золотого сечения. Оно встречается как в природе, так и в человеческой деятельности. Два знаменитых (и совершенно не связанных друг с другом) примера золотого сечения — описание формулы спиралевидной раковины наутилуса и механизм возврата к прежнему уровню цен на рынке.

Обратите внимание: веса из табл. 10.1 напоминают начальные элементы последовательности Фибоначчи. Вместо табл. 10.1 в качестве весов можно выбрать любые 4 последовательных числа из последовательности Фибоначчи (например, [89, 144, 233, 377]). Независимо от вашего выбора при использовании их для сети на рис. 10.4 риск всегда будет равен 0,64, потому что отношение весов остается равным φ. Если W_G — вес зеленых активностей, то другие веса равны:

$$W_Y = \varphi \times W_G$$
$$W_R = \varphi^2 \times W_G$$
$$W_C = \varphi^3 \times W_G,$$

а формула риска возникновения критичности может быть записана в виде:

$$\text{Риск} = \frac{\varphi^3 \times W_G \times N_C + \varphi^2 \times W_G \times N_R + \varphi \times W_G \times N_C + W_G \times N_G}{\varphi^3 \times W_G \times N}.$$

Так как W_G встречается во всех элементах числителя и знаменателя, уравнение можно упростить до следующего вида:

$$\text{Риск} = \frac{\varphi^3 \times N_C + \varphi^2 \times N_R + \varphi \times N_Y + N_G}{\varphi^3 \times N}.$$

С приближенным значением φ формула сокращается до следующего вида:

$$\text{Риск} = \frac{4{,}24 \times N_C + 2{,}62 \times N_R + 1{,}62 \times N_Y + N_G}{4{,}24 \times N}.$$

Я называю эту модель риска *моделью риска Фибоначчи*.

Значение риска Фибоначчи

Максимальное значение, которое может быть достигнуто по формуле риска Фибоначчи, — 1,0 в полностью критической сети. Минимальное достижимое значение равно 0,24 (1/4,24), чуть меньше минимального значения модели риска возникновения критичности 0,25 (при использовании набора [1, 2, 3, 4] в качестве весов).

Риск активности

В модели риска возникновения критичности используются широкие категории рисков. Например, если определить временной резерв, превышающий 25 дней, как зеленый, то две активности — с 30 и с 60 днями временного резерва — будут помещены в одну зеленую категорию и будут иметь одинаковое значение риска. Чтобы лучше учесть вклад в риск каждой отдельной активности, я создал *модель риска активности*. Такая модель обладает гораздо большей дискретностью, чем модель риска возникновения критичности.

Формула риска активности выглядит так:

$$\text{Риск} = 1 - \frac{F_1 + \ldots + F_i + \ldots + F_N}{M \times N} = 1 - \frac{\sum_{i=1}^{N} F_i}{M \times N},$$

где:

- F_i — временной резерв активности i;
- N — количество активностей в проекте;
- M — максимальный временной резерв любой активности в проекте, то есть $\text{Max}(F_1, F_2, \ldots, F_N)$.

Как и в случае с риском возникновения критичности, из анализа следует исключить активности с нулевой продолжительностью (контрольные точки и фиктивные активности).

Применение формулы риска активностей к сети на рис. 10.4 дает следующий результат:

$$\text{Риск} = 1 - \frac{30+30+30+30+10+10+5+5+5+5}{30 \times 16} = 0{,}67.$$

Значения риска активности

Модель риска активности не определена, если все активности являются критическими. Тем не менее в предельном случае для большой сети (большое зна-

чение N), которая содержит только одну некритическую активность с временным резервом M, модель достигает 1,0:

$$\text{Риск} \approx 1 - \frac{F_1}{M \times N} = 1 - \frac{M}{M \times N} = 1 - \frac{1}{N} \approx 1 - 0 = 1{,}0.$$

Минимальное значение риска активности равно 0, когда все активности в сети имеют одинаковые уровни временного резерва M:

$$\text{Риск} = 1 - \frac{\sum_{i=1}^{N} M}{M \times N} = 1 - \frac{M \times N}{M \times N} = 1 - 1 = 0.$$

Хотя риск активности теоретически может достигать 0, на практике вам вряд ли попадется такой проект, потому что все проекты всегда имеют ненулевое значение риска.

Потенциальная проблема при вычислении

Модель риска активности хорошо работает только тогда, когда временные резервы проектов более или менее равномерно распределены между наименьшим и наибольшим временным резервом в сети. Аномальное значение временного резерва, существенно превышающее другие временные резервы, внесет искажения в вычисления и породит завышенное значение риска. Например, представьте, что одногодичный проект содержит одну активность недельной продолжительности, которая может находиться где угодно между началом и концом проекта. Такая активность будет иметь почти год временного резерва, как показано на рис. 10.5.

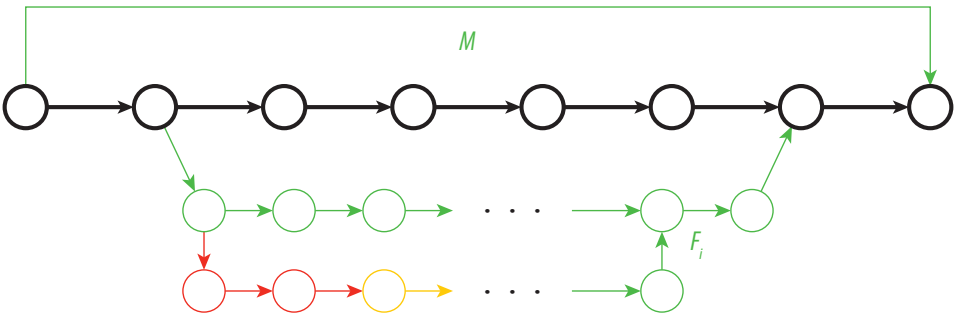


Рис. 10.5. Сеть, содержащая активность с аномально высоким временным резервом

На рис. 10.5 изображен критический путь (толстая линия) и многие активности с цветовым кодированием уровня временного резерва (F_i) внизу. Активность, изображенная над критически путем, коротка, но она имеет огромный временной резерв M.

Так как значение M много больше любого другого F_i, формула риска активности дает число, приближающееся к 1:

$$M \gg F_i$$

$$\text{Риск} = 1 - \frac{\sum_{i=1}^{N} F_i}{M \times N} \approx 1 - \frac{F_i \times N}{M \times N} \approx 1 - \frac{F_i}{M} \approx 1 - 0 = 1{,}0.$$

В следующей главе продемонстрирована эта ситуация, а также приводится простой и эффективный способ выявления и корректировки аномальных временных резервов.

Риск активности также дает заниженное значение риска активности, когда проект не содержит много активностей и временные резервы некритических активностей имеют сходные и даже одинаковые значения. Тем не менее если не считать редких и искусственных случаев, модель риска активности оценивает риск правильно.

Риск активности и риск возникновения критичности

Для реальных проектов сколько-нибудь серьезного размера модели риска возникновения критичности и риска активности дают очень похожие результаты. У каждой модели имеются свои достоинства и недостатки. В общем случае риск возникновения критичности лучше отражает человеческую интуицию, тогда как риск активности в большей степени приспособлен к различиям между отдельными активностями. Моделирование риска возникновения критичности часто требует калибровки или субъективных решений, но оно нечувствительно к тому, насколько равномерно распределены значения временных резервов. Риск активности чувствителен к присутствию аномально высоких значений, но он легко вычисляется и не требует особой калибровки. Корректировку аномальных значений временных резервов даже можно автоматизировать.

> **ПРИМЕЧАНИЕ** Если риск активности и риск возникновения критичности сильно различаются, вы должны определить причину. Возможно, калибровка риска возникновения критичности была проведена некорректно или риск активности был искажен из-за неравномерного распределения временных резервов. Если найти причину не удалось, используйте модель риска Фибоначчи в качестве модели риска для арбитража.

Уплотнение и риск

Как упоминалось ранее, риск слегка убывает с высоким уплотнением, что отражает наше интуитивное представление о том, что короткие проекты более безопасны. Числовое моделирование рисков предоставляет объяснение для этого явления. Единственный реальный способ сильного уплотнения программного проекта — введение параллельной работы. В главе 9 были перечислены некоторые идеи организации параллельной работы, такие как расщепление активностей и выполнение менее зависимых фаз параллельно с другими активностями или введение дополнительных активностей, которые делают возможной параллельную работу. На рис. 10.6 этот эффект изображен в количественном представлении.

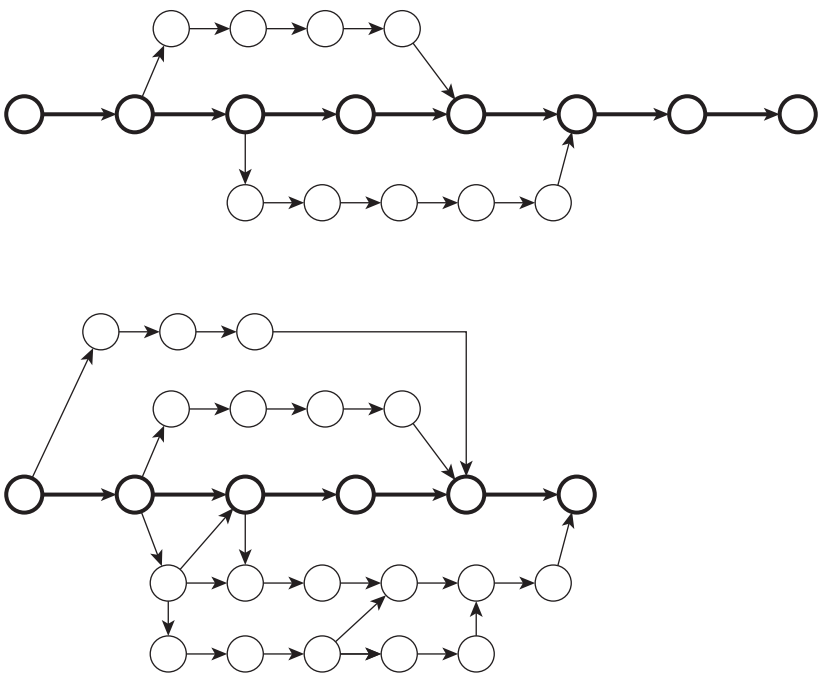


Рис. 10.6. С высоким уплотнением сеть становится более параллельной

На рис. 10.6 изображены две сети; нижняя диаграмма представляет собой сжатую версию верхней. Уплотненное решение содержит меньше критических активностей, имеет более короткий критический путь и больше некритических активностей, выполняемых параллельно. При измерении риска таких уплотненных проектов присутствие большего количества активностей с временным резервом и меньшим количеством критических активностей сократит величи-

ну риска, производимого как моделью риска возникновения критичности, так и моделью риска активности.

Риск выполнения

Хотя проектный риск для проекта с высоким параллелизмом может быть ниже, чем проектный риск решения с меньшим уплотнением, выполнение такого проекта может представлять большую трудность из-за дополнительных зависимостей и возросшего количества активностей, которые необходимо планировать и отслеживать. Такой проект будет иметь жесткие ограничения по срокам и потребует большей команды. В сущности, проект с высоким уплотнением преобразует проектный риск в риск выполнения. Хорошим признаком ожидаемого риска выполнения является сложность сети. В главе 12 вы узнаете, как выразить сложность выполнения в числовой форме.

Разуплотнение риска

При уплотнении проекта риск с большой вероятностью возрастет, причем обратное также истинно (до определенного момента): при ослаблении сроков проекта можно понизить его риск. Я называю этот метод *разуплотнением риска*. Вы намеренно проектируете проект на более позднюю реализацию, чтобы ввести временной резерв на критическом пути. Разуплотнение риска — лучший способ сократить непрочность проекта, его чувствительность к непредвиденным обстоятельствам.

Проекты следует разуплотнять тогда, когда доступные решения слишком рискованны. Другие причины для разуплотнения проекта — беспокойство о текущих перспективах из-за плохой истории, слишком большое количество неизвестных или нестабильная среда, которая постоянно изменяет приоритеты и ресурсы.

Как объяснялось в главе 7, классическая ошибка при попытке сокращения риска — завышение и занижение оценок. Это только ухудшает ситуацию и снижает вероятность успеха. Вся суть разуплотнения — сохранение исходных оценок без изменений и повышение временного резерва на всех сетевых путях.

В то же время нужно знать меру с разуплотнением. Используя модели рисков, можно измерить эффект разуплотнений и остановиться при достижении целевого значения разуплотнения (см. далее в этом разделе). Если все активности имеют высокий временной резерв, избыточное разуплотнение будет иметь низкую эффективность. Все дополнительное разуплотнение за пределами этой точки не сократит проектный риск, но только увеличит общую переоценку риска и приведет к потере времени.

Разуплотнение может применяться к любому плану проекта, хотя обычно оно применяется только к нормальному решению. Разуплотнение отчасти смещает проект в нерентабельную зону (см. рис. 10.2) с увеличением времени и затрат проекта. Когда вы разуплотняете план проекта, вы все равно планируете его с исходным комплектованием. Не поддавайтесь искушению потребить дополнительный временной резерв и сократить комплектование — это противоречит исходной цели разуплотнения риска.

Как выполняется разуплотнение

Прямолинейный способ разуплотнения проекта заключается в том, чтобы сместить последнюю активность или последнее событие в проекте по временной шкале. При этом временной резерв будет добавлен во все предшествующие активности сети. В случае сети на рис. 10.4 разуплотнение активности 16 на 10 дней приводит к риску возникновения критичности 0,47 и риску активности 0,52. Разуплотнение активности 16 на 30 дней приведет к риску возникновения критичности 0,3 и риску активности 0,36.

Более сложный метод заключается в том, чтобы также разуплотнить одну или две ключевые активности на критическом пути (например, активность 8 на рис. 10.4). В общем случае чем ниже находится разуплотняемая сеть, тем больше необходимо разуплотнять, потому что любая неудача в предшествующей активности может привести к потреблению временного резерва последующих активностей. Чем ранее в сети выполняется разуплотнение, тем ниже вероятность того, что весь введенный временной резерв будет потреблен.

Цель разуплотнения

При разуплотнении проекта следует стремиться к проведению разуплотнения до тех пор, пока риск не упадет до 0,5. На рис. 10.7 показана эта точка на идеальной кривой риска с использованием логистической функции с асимптотическим приближением к 1 и 0.

Если проект имеет очень короткую продолжительность, значение риска почти достигает 1,0 и риск максимизируется. В это время кривая риска почти горизонтальна. Поначалу добавление времени в проект не приводит к значительному снижению риска. При добавлении большего времени в какой-то момент кривая риска начинает опускаться, и чем больше времени будет выделено для проекта, тем круче изгибается кривая. Но даже с еще большим временем кривая риска становится более пологой, предоставляя меньшее сокращение риска за дополнительное время. Точка, в которой кривая риска имеет наибольшую крутизну, является точкой с наивысшей эффективностью разуплотнения (то есть наибольшего сокращения риска при наименьшей величине разуплотнения). Эта точка определяет цель разуплотнения риска. Так как логистиче-

ская функция на рис. 10.7 является симметричной кривой между 0 и 1, точка перегиба достигается при значении риска 0,5.

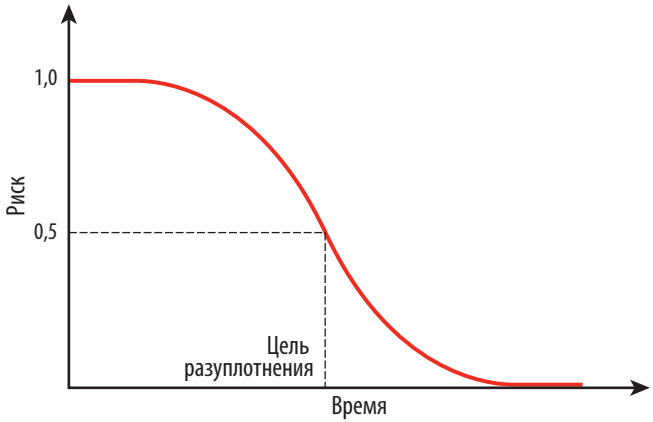


Рис. 10.7. Цель разуплотнения на идеальной кривой риска

Чтобы определить, как цель разуплотнения связана с затратами, сравните реальную кривую риска с кривой прямых затрат (рис. 10.8). Реальная кривая риска ограничена более узким диапазоном, чем идеальная кривая риска, и никогда не достигает ни 0, ни 1, хотя и ведет себя аналогично логистической функции между значениями максимума и минимума. Как упоминалось в начале этой главы, точка кривой риска с наибольшей крутизной (в которой

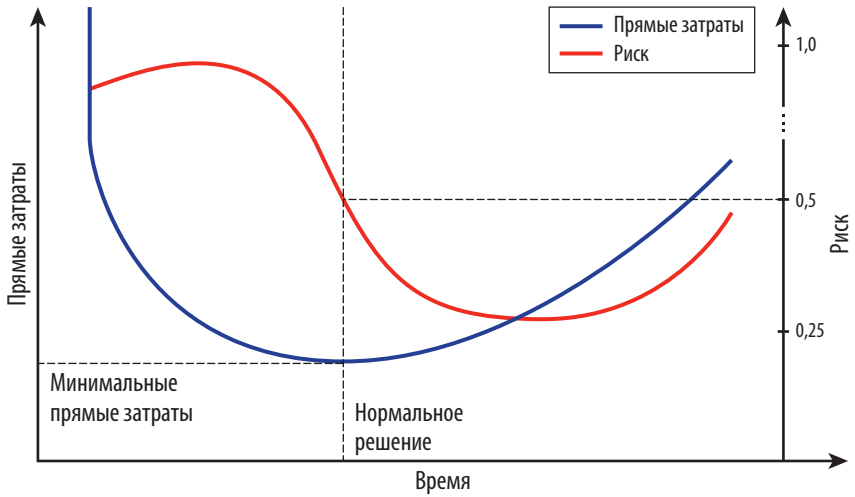


Рис. 10.8. Точка минимальных затрат совпадает с точкой риска 0,5

вогнутая кривая становится выпуклой) находится в точке минимальных прямых затрат, которая совпадает с целевым значением разуплотнения (см. рис. 10.8).

Так как риск продолжает уменьшаться справа от 0,5, значение 0,5 можно считать минимальной целью разуплотнения. Как и прежде, вы должны отслеживать поведение кривой риска и избегать чрезмерного разуплотнения.

Если точка минимальных прямых затрат проекта также является наилучшей в отношении риска, это делает ее оптимальной точкой планирования проекта, обеспечивающей наименьшие прямые затраты с лучшим риском. Эта точка не слишком рискованна, но и не слишком безопасна, то есть обеспечивает наибольший возможный эффект от добавления времени в проект.

> **ПРИМЕЧАНИЕ** Теоретически точка минимальных прямых затрат проекта совпадает с нормальным решением и точкой наибольшей крутизны кривой риска. На практике это редко бывает так, потому что модель по своей основе является дискретной, и обычно приходится делать поправки на реальность. Возможно, ваше нормальное решение близко к точке минимальных прямых затрат, но не совпадает с ней. Это означает, что нормальное решение часто приходится уплотнять до точки перегиба кривой риска.

Метрики риска

В завершение этой главу приведу несколько легко запоминающихся метрик и эмпирических правил. Как и другие метрики планирования, они должны использоваться скорее как рекомендации. Нарушение метрики является тревожным сигналом, и вы всегда должны исследовать причину происходящего.

- **Выдерживайте значения риска в диапазоне от 0,3 до 0,75.** Ваш проект никогда не должен достигать экстремальных значений риска. Очевидно, значение риска 0 или 1,0 является бессмысленным. Риск не должен быть слишком низким: так как модель риска возникновения критичности не может упасть ниже 0,25, можно округлить нижнюю границу 0,25 до 0,3 как нижнюю границу для любого проекта. При уплотнении проекта задолго до того, как риск поднимется до 1,0 (полностью критичный проект), уплотнение следует остановить. Даже значение риска 0,9 или 0,85 все равно остается высоким. Если нижняя четверть от 0 до 0,25 недоступна, ради симметрии также следует избегать верхней четверти значений рисков от 0,75 до 1,0.
- **Проводите уплотнение до 0,5.** Идеальная цель сжатия — риск 0,5, так как она соответствует точке перегиба на кривой риска.

- **Избегайте чрезмерного уплотнения.** Как обсуждалось ранее, уплотнение за пределами целевого значения теряет эффективность, а чрезмерное уплотнение приводит к повышению риска.
- **Поддерживайте нормальные решения на уровне ниже 0,7.** Хотя возросший риск может стать ценой, которую вы платите за уплотненное решение, для нормальных решений он нежелателен. Вернемся к аргументу о симметричности: если риск 0,3 является нижней границей для всех решений, риск 0,7 должен быть верхней границей для нормального решения. Нормальные решения с высоким риском всегда должны уплотняться.

Моделирование рисков и метрики рисков должны стать составной частью планирования проекта. Постоянно измеряйте риски, чтобы определить, в каком состоянии находится проект и куда он движется.

11 Планирование проекта в действии

Трудность, с которой сталкиваются многие начинающие проектировщики при планировании проекта, — не конкретные приемы и концепции планирования, а скорее процесс планирования от начала до конца. Также легко увязнуть в деталях и забыть о конечной цели работы. Без практического опыта вам будет трудно, когда вы столкнетесь с первой проблемой или ситуацией, в которой все идет не так, как ожидалось. Было бы нереально пытаться рассмотреть все возможные ситуации и ваши реакции на них. Вместо этого лучше освоить процесс мышления, который должен применяться в процессе планирования проекта.

В этой главе процесс и образ мышления продемонстрированы на подробном описании работы по планированию. Особое внимание уделяется систематическому анализу шагов и итераций. Вы увидите многочисленные наблюдения и эмпирические правила, научитесь переключаться между вариантами планирования проекта, узнаете, как сосредоточиться на том, что имеет смысл для ваших целей, и научитесь оценивать плюсы и минусы разных компромиссных решений. В процессе изложения материала эта глава демонстрирует практическое применение идей из предыдущих глав, а также синергию, которая достигается объединением методов планирования проекта. В ней также рассматриваются такие аспекты планирования проектов, как исходные предпосылки планирования, понижение сложности, комплектование и формирование графика, введение ограничений, уплотнение, риск и планирование. Эта глава должна научить вас применению процесса и приемов планирования (вместо простого рассмотрения реального примера).

Формулировка миссии

Ваша задача заключается в планировании проекта для построения типичной бизнес-системы. Система была спроектирована с применением Метода, но в этой главе данный факт роли не играет. В общем случае входные данные для работы по планированию проекта должны включать следующие компоненты:

- **Статическая архитектура.** Статическая архитектура используется для создания исходного списка активностей, непосредственно связанных с программированием.
- **Цепочки вызовов или диаграммы последовательностей.** Чтобы построить цепочку вызовов или диаграмму последовательности, проанализируйте сценарии использования и пути их распространения в системе. Таким образом вы получите черновой вариант структурных зависимостей между активностями.
- **Список активностей.** В список включаются все активности — как непосредственно связанные с программированием, так и другие.
- **Оценка продолжительности.** Для каждой активности вы даете точную оценку ее продолжительности (и задействованных ресурсов) или работаете с коллегами для получения оценок.
- **Предпосылки планирования.** Вы фиксируете свои предположения относительно комплектования, доступности, времени выхода на рабочий режим, технологии, качества и т. д. Обычно создаются несколько таких наборов предположений, при этом каждый набор дает отдельный вариант планирования проекта.
- **Некоторые ограничения.** Запишите все явно заданные ограничения. Также следует включить возможные или вероятные ограничения и вести планирование соответствующим образом. В этой главе приведено несколько примеров обработки ограничений.

Статическая архитектура

На рис. 11.1 изображена статическая архитектура системы. Как нетрудно догадаться, система имеет относительно небольшие размеры. В нее включены два *Клиента*, пять компонентов бизнес-логики, три компонента *Доступ к ресурсу*, два *Ресурса* и три компонента *Вспомогательные средства*.

Хотя система на рис. 11.1 была построена по образцу реальной системы, достоинства этой конкретной архитектуры в этой главе нас не интересуют. При планировании проекта следует избегать превращения планирования проекта в анализ системной архитектуры. Даже неудачная архитектура должна иметь адекватный план проекта, максимизирующий вероятность соблюдения обязательств по проекту.

Цепочки вызовов

Система имеет только два базовых сценария использования и две цепочки вызовов. Первая цепочка вызовов, показанная на рис. 11.2, завершается публикацией события. Вторая цепочка вызовов на рис. 11.3 представляет обработку события подписчиками.

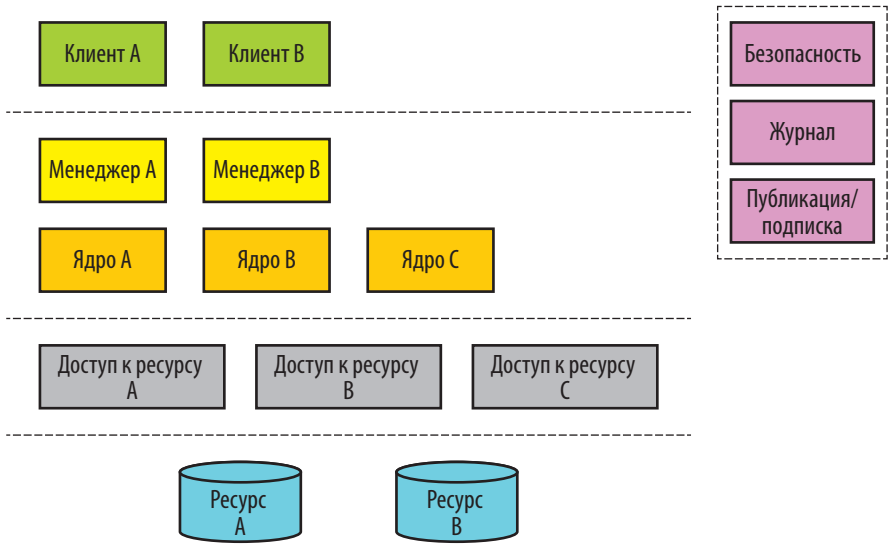


Рис. 11.1. Статическая архитектура системы

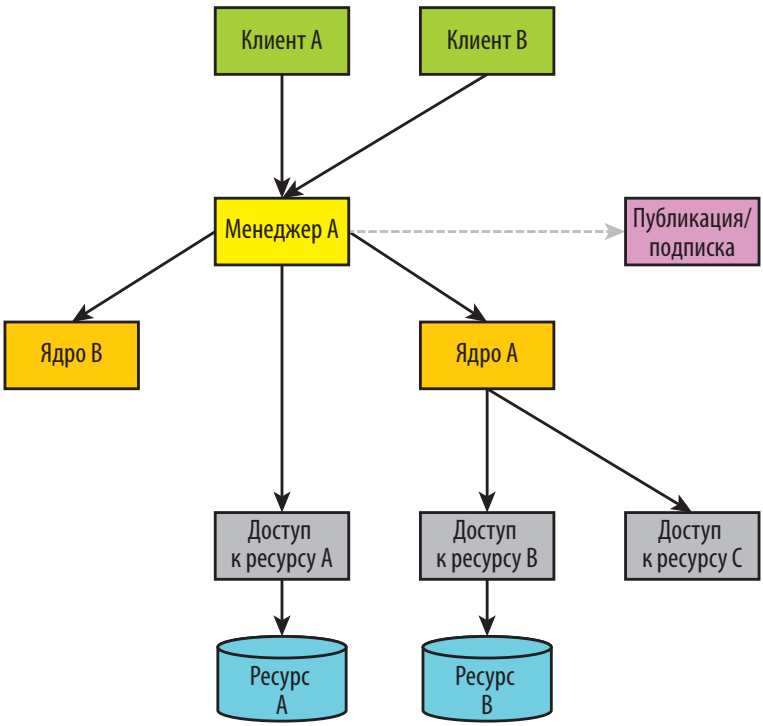


Рис. 11.2. Цепочка вызовов 1

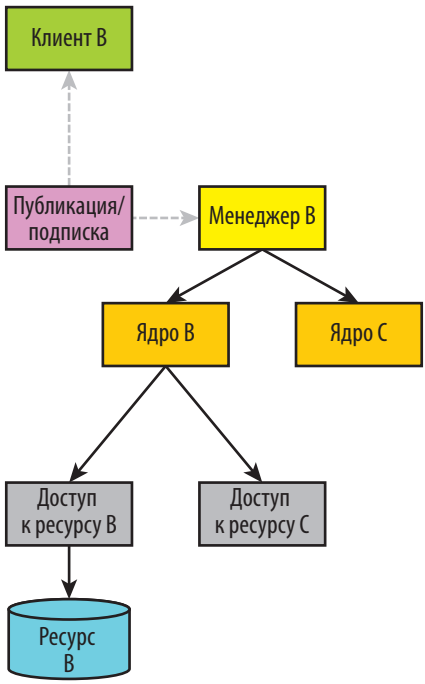


Рис. 11.3. Цепочка вызовов 2

Диаграмма зависимостей

Вы должны проанализировать цепочки вызовов и построить черновую версию зависимостей между компонентами архитектуры. Начните со стрелок, соединяющих компоненты (независимо от транспорта или механизма взаимодействий); каждая стрелка рассматривается как зависимость. Каждая зависимость должна учитываться ровно один раз. Тем не менее обычно диаграммы цепочек вызовов не представляют всей картины, потому что на них часто опускаются неявные зависимости. В данном случае все компоненты архитектуры (кроме *Ресурсов*) зависят от *Журнала*, а *Клиенты* и *Менеджеры* зависят от компонента *Безопасность*. На основе этой дополнительной информации можно нарисовать диаграмму зависимостей, изображенную на рис. 11.4.

Как видно из диаграммы, даже в простой системе всего с двумя сценариями использования диаграмма зависимостей получается громоздкой и неудобной для анализа. Простой метод, которым можно воспользоваться для сокращения сложности, — устранение зависимостей, дублирующих унаследованные зависимости. Унаследованные зависимости обусловлены транзитивными за-

висимостями[1] — зависимостями, которые неявно наследуются активностями, зависящими от других активностей.

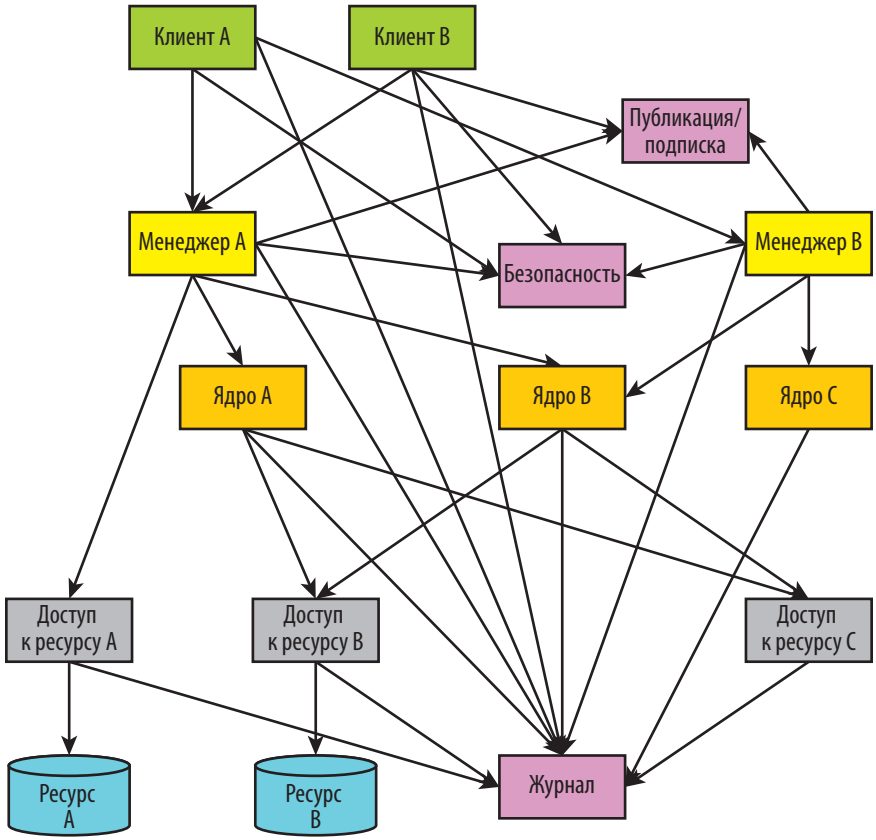


Рис. 11.4. Исходная диаграмма зависимостей

На рис. 11.4 *Клиент A* зависит от *Менеджера A* и компонента *Безопасность*; *Менеджер A* также зависит от компонента *Безопасность*. Это означает, что зависимость между *Клиентом A* и компонентом *Безопасность* можно опустить. Используя унаследованные зависимости, можно сократить рис. 11.4 до рис. 11.5.

[1] https://en.wikipedia.org/wiki/Transitive_dependency

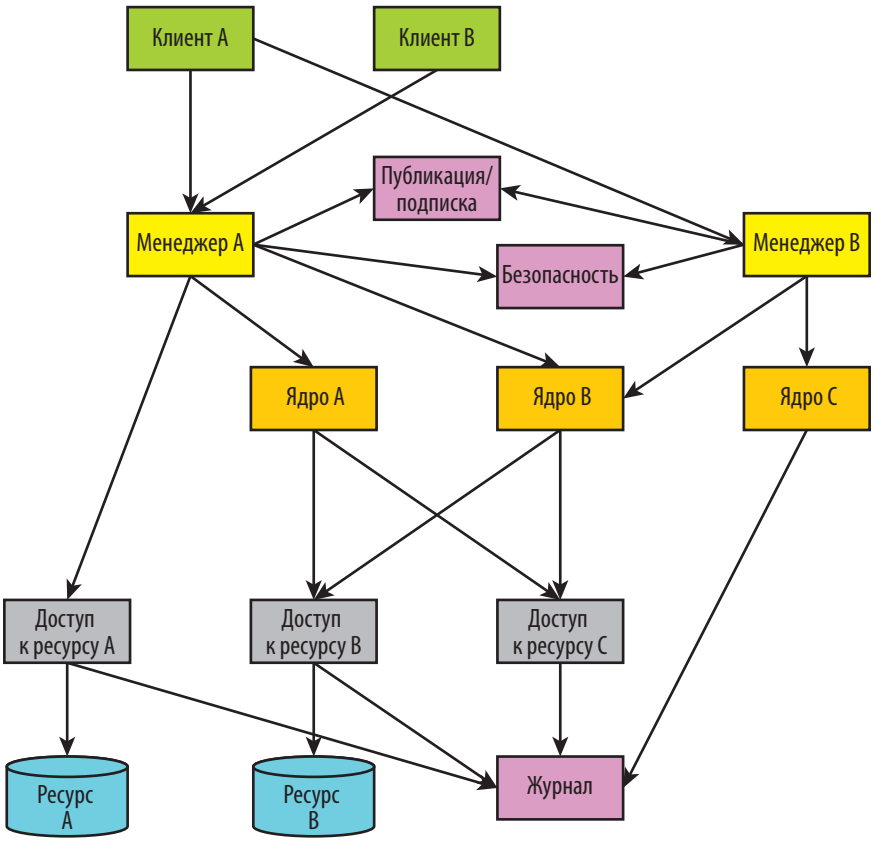


Рис. 11.5. Диаграмма зависимостей после консолидации унаследованных зависимостей

Список активностей

Бесспорно, диаграмма на рис. 11.5 проще рис. 11.4, но этого недостаточно — она все еще имеет слишком структурную природу и на ней отображаются только активности, связанные с программированием. Вы должны построить подробный список всех активностей в проекте. В данном случае список активностей, не связанных с программированием, включает дополнительную работу по требованиям, архитектуре (например, проверка технологии или демонстрационный сервис), план проекта, план тестирования, тестовую оснастку и тестирование системы. В табл. 11.1 перечислены все активности проекта, оценка их продолжительности и их зависимости от предшествующих активностей.

Таблица 11.1. Активности, продолжительность и зависимости

ID	Активность	Продолжительность (дни)	Зависит от
1	Требования	15	
2	Архитектура	20	1
3	Планирование проекта	20	2
4	План тестирования	30	3
5	Тестовая оснастка	35	4
6	Журнал	15	3
7	Безопасность	20	3
8	Публикация/подписка	5	3
9	Ресурс A	20	3
10	Ресурс B	15	3
11	Доступ к ресурсу A	10	6,9
12	Доступ к ресурсу B	5	6,10
13	Доступ к ресурсу C	15	6
14	Ядро A	20	12,13
15	Ядро B	25	12,13
16	Ядро C	15	6
17	Менеджер A	20	7,8,11,14,15
18	Менеджер B	25	7,8,15,16
19	Клиентское приложение 1	25	17,18
20	Клиентское приложение 2	35	17
21	Тестирование системы	30	5,19,20

Диаграмма сети

Располагая списком активностей и зависимостей, можно нарисовать сеть проекта в виде стрелочной диаграммы. На рис. 11.6 изображена исходная диаграмма сети. Числа на диаграмме соответствуют идентификаторам активностей в табл. 11.1. Жирными линиями и числами выделен критический путь.

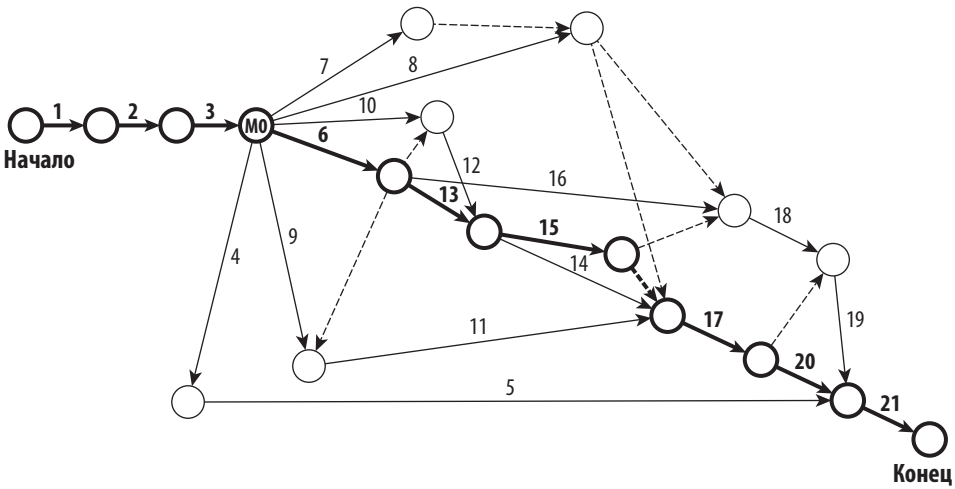


Рис. 11.6. Исходная диаграмма сети

О контрольных точках

Как было указано в главе 8, *контрольная точка* представляет собой событие в проекте, обозначающее завершение значительной части проекта, включая основные усилия по интеграции. Даже на столь ранней стадии планирования проекта необходимо назначить событие, завершающее Планирование проекта (активность 3) контрольной точкой M0 анализа SDP. В данном случае M0 является завершением начальной стадии (краткое название нечеткой начальной стадии) проекта, включающей проработку требований, архитектуры и планирование проекта. В результате анализ SDP становится явной частью плана. Контрольные точки могут находиться на критическом пути или вне его, они могут быть приватными или открытыми. Открытые контрольные точки становятся признаками прогресса для руководства и заказчиков, приватные контрольные точки — внутренние метки для команды. Если контрольная точка находится вне критического пути, желательно сделать ее приватной, потому что она может переместиться в результате задержки где-то в предшествующих активностях. На критическом пути контрольные точки могут быть как приватными, так и открытыми; они напрямую коррелируют с выполнением обязательств по проекту в отношении времени и затрат. Другое применение контрольных точек — создание зависимости даже в том случае, если цепочки вызовов не определяют такую зависимость. Такой контрольной точкой является анализ SDP — никакие активности, направленные на построение системы, не должны начинаться до анализа SDP. Такие контрольные точки форсированных зависимостей также упрощают сеть, как вскоре будет показано на другом примере.

Исходная продолжительность

Сеть активностей, перечисленных в табл. 11.1, можно построить в программе планирования проекта, которая даст первое представление о продолжительности проекта. Для этого проекта она равна 9,0 месяца. Тем не менее без назначения ресурсов определить затраты проекта невозможно.

> **ПРИМЕЧАНИЕ** В состав файлов, прилагаемых к книге, включены файлы Microsoft Project и электронные таблицы Excel для всех комбинаций и итераций, представленных в этой главе. В тексте главы приводятся только объяснения и сводная информация.

Предпосылки планирования

Чтобы продолжить планирование, следует составить список предположений планирования, особенно требований комплектования. Список должен выглядеть примерно так:

- Один менеджер проекта на все время проекта.
- Один менеджер продукта на все время проекта.
- Один архитектор на все время проекта.
- Один разработчик на сервис для любой активности, связанной с программированием. После того как сервис будет завершен, разработчик может переключиться на другую активность.
- Один архитектор базы данных для каждого из *Ресурсов*. Эта работа не зависит от разработки кода и может выполняться параллельно.
- Один тестировщик от начала построения сервисов системы до конца тестирования.
- Один дополнительный тестировщик на время тестирования системы.
- Один инженер по тестированию на активности плана тестирования и тестовой оснастки.
- Один специалист DevOps от начала построения до конца тестирования.

По сути, этот список содержит перечень ресурсов, необходимых для завершения проекта. Также обратите внимание на структуру списка: «Один X для Y». Если вы не можете представить необходимый уровень комплектования в таком виде, вероятно, вы не понимаете собственные требования к комплектованию или упускаете какое-то ключевое предположение из области планирования.

Также вы должны сделать два дополнительных предположения о разработчиках на время тестирования и время бездействия. Во-первых, в этом конкретном проекте разработчики будут выполнять свою работу с таким качеством, что во

время тестирования системы их участие не потребуется. Во-вторых, разработчики между активностями считаются прямыми затратами. Строго говоря, время бездействия должно учитываться как косвенные затраты, потому что оно не связано с активностями проекта, однако проект должен его оплачивать. Тем не менее многие руководители проектов стараются назначить бездействующим разработчикам некоторые активности в поддержку других активностей разработки, даже если это означает, что на один сервис будут временно назначены сразу несколько разработчиков. С учетом этого предположения разработчики между активностями продолжают учитываться как прямые затраты.

Фазы проекта

Каждая активность в проекте всегда принадлежит некоторой фазе (или типу активностей). К числу типичных фаз относятся начальная стадия, проектирование, инфраструктура, сервисы, UI, тестирование, развертывание и т. д. Фаза может содержать любое количество активностей, причем активности в фазе могут перекрываться по времени. Менее очевидно то, что фазы непоследовательны, что они сами могут перекрываться и даже запускаться и останавливаться. Самый простой способ планирования фаз — преобразование списка предположений планирования в таблицу «роль/фаза»; пример приведен в табл. 11.2.

Таблица 11.2. Роли и фазы

Роль	Начальная стадия	Инфра-структура	Сервисы	Тестирование
Архитектор	X	X	X	X
Менеджер проекта	X	X	X	X
Менеджер продукта	X	X	X	X
DevOps		X	X	X
Разработчики		X	X	
Тестировщики			X	X

Аналогичным образом можно добавить другие роли, необходимые на протяжении всей фазы, например эксперты по UX (опыт взаимодействия с пользователем) или безопасности. Тем не менее не следует включать роли, необходимые только для конкретных активностей (например, инженер по тестированию).

Таблица 11.2 является примитивной формой представления распределения комплектования. Отношения между ролями и фазами исключительно важны при построении диаграммы распределения комплектования, потому что вы должны учитывать использование всех ресурсов независимо от того, назначены ли они конкретным активностям проекта. Например, в табл. 11.2 присутствие архитектора требуется на протяжении всего проекта. В свою очередь,

в диаграмме распределения комплектования архитектор отображается на протяжении всего проекта. Таким образом можно учитывать все ресурсы, необходимые для построения правильной диаграммы распределения комплектования и вычисления затрат.

> **ПРИМЕЧАНИЕ** Вряд ли вы будете получать предположения планирования в готовом виде, как в этой главе. В начальной стадии проекта всегда происходит некоторый анализ и попытки согласования, когда вы пытаетесь выделить конкретные предположения планирования. Также можно действовать в обратном направлении: сначала сформулируйте предположения планирования и распределения комплектования в том виде, который предложен здесь, а затем запросите у коллег обратную связь и комментарии.

Поиск нормального решения

Имея список активностей, зависимостей и предположений планирования, переходите к итеративному поиску нормального решения. На первом проходе предположите, что в вашем распоряжении имеются неограниченные ресурсы, но используйте столько ресурсов, сколько требуется для беспрепятственного продвижения по критическому пути. Таким образом формируется наименее ограниченный вариант построения системы с наименьшим уровнем ресурсов.

Неограниченные ресурсы (итерация 1)

Изначально также предполагается, что комплектование обладает неограниченной эластичностью. Также можно внести незначительные поправки на реальность (хотя, возможно, они не понадобятся). Например, нет смысла нанимать человека на одну неделю, если можно потратить часть временного резерва, чтобы обойтись без этого ресурса. Предполагается, что каждый конкретный набор навыков будет доступен тогда, когда потребуется. С этими предположениями должна быть получена та же продолжительность проекта, как и до назначения ресурсов. В самом деле, после комплектования проекта подобным образом продолжительность остается равной 9,0 месяца, и вы получаете диаграмму планируемой освоенной ценности, изображенную на рис. 11.7. На этой диаграмме видна общая форма пологой S-образной кривой, но она не настолько плавная, как должна быть. В соответствии с процессом, описанным в главе 7, на рис. 11.8 показана соответствующая диаграмма распределения комплектования проекта. В плане используются четыре разработчика, два архитектора базы данных и один инженер по тестированию и не потребляется никакой временной резерв. Вычисленная величина общих затрат по проекту составляет 58,3 человеко-месяца.

Вспомните, о чем говорилось в главе 9: поиск нормального решения является итеративным процессом (см. рис. 9.8) просто из-за того, что наименьший

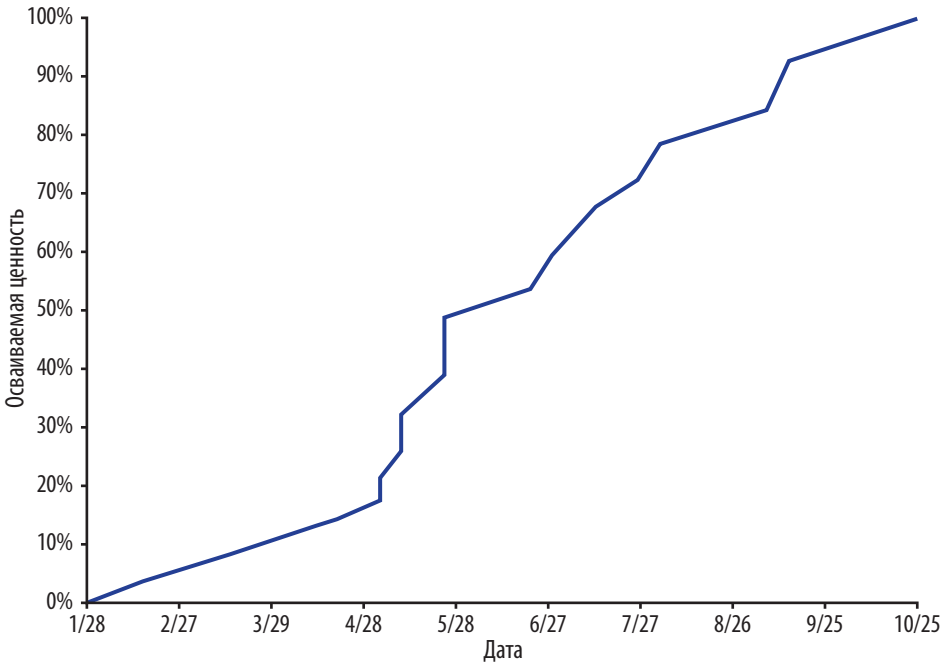


Рис. 11.7. Планируемая осваиваемая ценность с неограниченными ресурсами

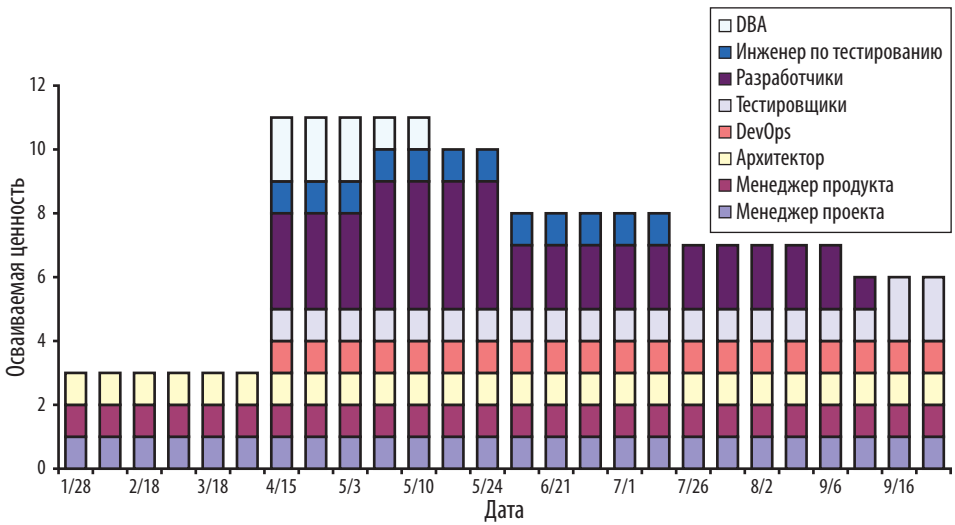


Рис. 11.8. Распределение комплектования с неограниченными ресурсами

уровень комплектования неизвестен на момент начала планирования. Следовательно, первый набор результатов еще не является нормальным решением. На следующей итерации вы учитываете влияние реальности, потребляете временной резерв для снижения нестабильности комплектования, исправляете все очевидные дефекты в планировании и сокращаете сложность, насколько это возможно.

Сеть и проблемы с ресурсами

Первая итерация нормального решения страдает от нескольких ключевых проблем. Сначала предполагаются неограниченные и доступные ресурсы, включая ресурсы со специальными навыками. Очевидно, на практике ресурсы не могут быть неограниченными, а специальные навыки достаточно редки. Во-вторых, на диаграмме планируемого распределения комплектования (см. рис. 11.8) присутствует тревожный признак (упоминавшийся в главе 7), а именно резкий рост на входе в проект. Такое поведение вполне ожидаемо из-за предположений о доступности и эластичности комплектования. В-третьих, на протяжении реализации проект задействует некоторые ресурсы только на короткие периоды времени. Это может создать проблемы с точки зрения доступности и времени, необходимого на интеграцию новых работников. Возможно, вы планируете решить эту проблему за счет привлечения субподрядчиков, однако не стоит создавать проблемы, которые потом приходится решать. Распределение комплектования должно быть плавным, на графике не должно быть резких подъемов и падений. Создавайте другие разновидности проектов с ограничениями по ресурсам и более реалистичной эластичностью комплектования. Часто это приводит к сглаживанию как диаграммы распределения комплектования, так и диаграммы планируемой осваиваемой ценности.

Последняя проблема с этим решением — интеграционное давление на сервисы *Менеджер*. Из табл. 11.1 и диаграммы сети на рис. 11.6 видно, что *Менеджеры* (активности 17 и 18) должны интегрироваться с четырьмя или пятью другими сервисами. В идеале интеграция должна проводиться с одним-двумя сервисами за раз. Одновременная интеграция более двух сервисов с большой вероятностью приведет к нелинейному возрастанию сложности, потому что любые проблемы между сервисами будут накладываться друг на друга. Проблема усугубляется тем, что интеграция выполняется ближе к концу проекта, когда у вас уже не остается свободного времени для исправления ошибок.

Инфраструктура в начале (итерация 2)

Распространенный метод упрощения проекта — перемещение инфраструктурных сервисов (таких как *Журнал*, *Безопасность* или *Публикация/подписка*, а также и дополнительных инфраструктурных активностей, например автома-

тизации сборки) в начало проекта независимо от их естественных зависимостей в сети. Другими словами, сразу же после M0 разработчики будут работать над этими инфраструктурными сервисами. Вы даже можете добавить контрольную точку M1, которая обозначает момент завершения инфраструктуры, чтобы все остальные сервисы зависели от M1, как показано в подсети на рис. 11.9.

Завершение инфраструктуры на ранней стадии сокращает сложность сети (уменьшает количество зависимостей и соединительных линий) и снижает нагрузку, связанную с интеграцией, на *Менеджеров*. Такое переопределение исходных зависимостей обычно сокращает потребность в исходном комплектовании, потому что ни один из других сервисов не может запуститься до завершения M1. Кроме того, оно сокращает нестабильность комплектования и обычно приводит к более плавному распределению комплектования и более пологому наращиванию комплектования в начале проекта.

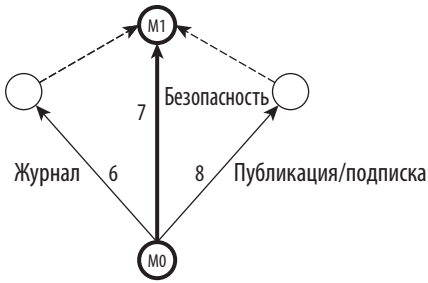


Рис. 11.9. Инфраструктура в начале

Другое важное преимущество разработки инфраструктуры в самом начале — ранняя доступность ключевых компонентов инфраструктуры. Это позволяет разработчикам интегрировать свою работу с инфраструктурой в процессе построения системы, вместо того чтобы достраивать и тестировать инфраструктурные сервисы (такие, как *Журнал* или *Безопасность*) после того, как все будет сделано. Доступность инфраструктурных сервисов до завершения бизнес-компонентов (*Доступ к ресурсу*, *Ядра*, *Менеджеры* и *Клиенты*) почти всегда желательна, даже если необходимость в этом неочевидна на первый взгляд.

Разработка инфраструктуры в начале изменяет исходное комплектование до трех разработчиков (по одному на сервис) до точки M1, в которой проект может принять четвертого разработчика (обратите внимание: вы все еще работаете над планом комплектования с неограниченными ресурсами). При повторении предыдущих шагов план с начальным созданием инфраструктуры увеличивает срок на 3% до 9,2 месяца и увеличивает общие затраты на 2% до 59 человеко-месяцев. В обмен на небольшие дополнительные затраты и время проект получает ранний доступ к ключевым сервисам и более простой, более

реалистичный план. Обновленный проект станет отправной точкой для следующей итерации.

Ограниченность ресурсов

Ресурсы, которые вы запрашиваете, не всегда доступны в тот момент, когда вы их запросили, поэтому будет разумно планировать меньший объем ресурсов (по крайней мере изначально) для снижения этого риска. Как поведет себя проект, если три разработчика будут недоступны в начале проекта? Разработкой инфраструктуры может заняться архитектор, или же проект может привлечь субподрядчиков: сервисы инфраструктуры не требуют знания предметной области, поэтому они становятся хорошими кандидатами для таких внешних и доступных ресурсов. Если в начале доступен только один разработчик, то этот разработчик может работать над всеми инфраструктурными компонентами последовательно. А может быть, изначально доступен только один разработчик, а второй разработчик сможет присоединиться к первому после того, как первая активность будет завершена.

Инфраструктура в начале с ограниченными ресурсами (итерация 3)

Выбрав последний, отчасти промежуточный, сценарий с одним, а затем с двумя разработчиками, пересчитайте проект и посмотрите, как проект поведет себя при ограниченных ресурсах. Вместо трех параллельных активностей (одна критическая) в начале, как на рис. 11.9, мы теперь имеем одну критическую активность, за которой идут две активности параллельно (одна критическая). Такая последовательная реализация активностей повышает продолжительность проекта. В этом варианте дедлайн повышается на 8% до 9,9 месяца, а общие затраты — на 4% до 61,5 человеко-месяца. На рис. 11.10 изображена полученная диаграмма распределения комплектования. Обратите внимание на постепенный ввод в проект разработчиков: сначала одного, затем двух и четырех.

Расширение критического пути за счет ограничения ресурсов также повышает временной резерв некритических активностей, охватывающих эту часть сети. По сравнению с неограниченными ресурсами временной резерв *Плана тестирования* (активность 4 на рис. 11.6) и *Тестовой оснастки* (активность 5 на рис. 11.6) повышается на 30%, временной резерв *Ресурса A* (активность 9 на рис. 11.6) повышается на 50%, а временной резерв *Ресурса B* (активность 10 на рис. 11.6) повышается на 100%. Эти изменения заслуживают внимания, потому что незначительное на первый взгляд изменение доступности ресурса привело к серьезному повышению временного резерва. Учтите, что это палка о двух концах: даже невинное на первый взгляд изменение может привести к краху временных резервов и неудаче проекта.

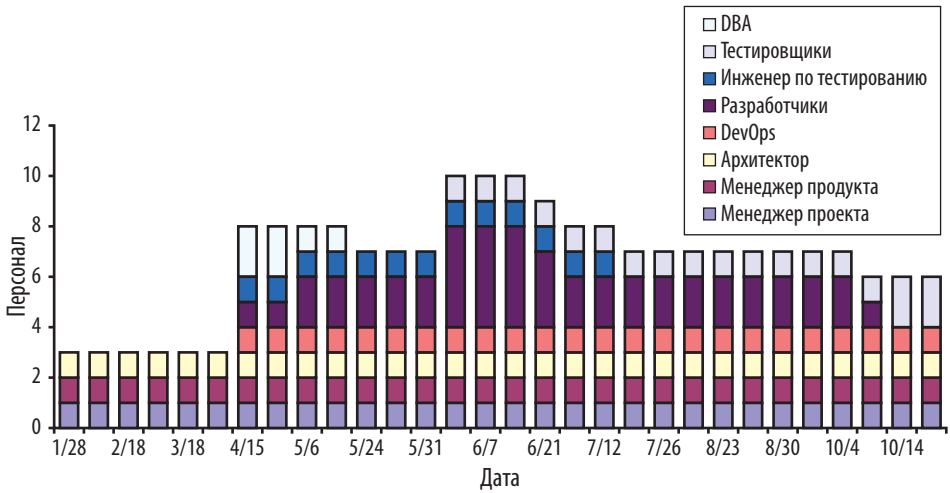


Рис. 11.10. Распределение комплектования с ограниченными ресурсами

Отсутствие архитекторов базы данных (итерация 4)

Помимо ограничения исходной доступности разработчиков, допустим, что проект не получил архитекторов базы данных, которые требовались в предыдущих решениях. Несомненно, такой сценарий следует признать реалистичным — найти такие квалифицированные ресурсы часто бывает достаточно сложно. В таком случае разработчики проектируют базы данных в меру своих способностей. Чтобы увидеть, как проект отреагирует на это новое ограничение, вместо того чтобы просто добавлять разработчиков, ограничьте проект четырьмя разработчиками (возможность увеличения количества разработчиков была бы равнозначна добавлению архитекторов базы данных). Как ни странно, это не изменяет продолжительность и результаты с общими затратами 62,7 человеко-месяца (возрастание всего на 2%). Это объясняется тем, что те же четыре разработчика начинают работать раньше и им даже не приходится потреблять временной резерв.

Дальнейшее ограничение ресурсов (итерация 5)

Так как проект может легко обойтись четырьмя разработчиками, следующий план с ограниченными ресурсами ограничивает количество доступных разработчиков тремя. Это также не изменяет продолжительность проекта, потому что четвертого разработчика можно частично компенсировать временным резервом. Что касается затрат, происходит 3-процентное снижение затрат до 61,1 человеко-месяца вследствие более эффективного использования разработчиков. На рис. 11.11 изображена полученная диаграмма распределения комплектования.

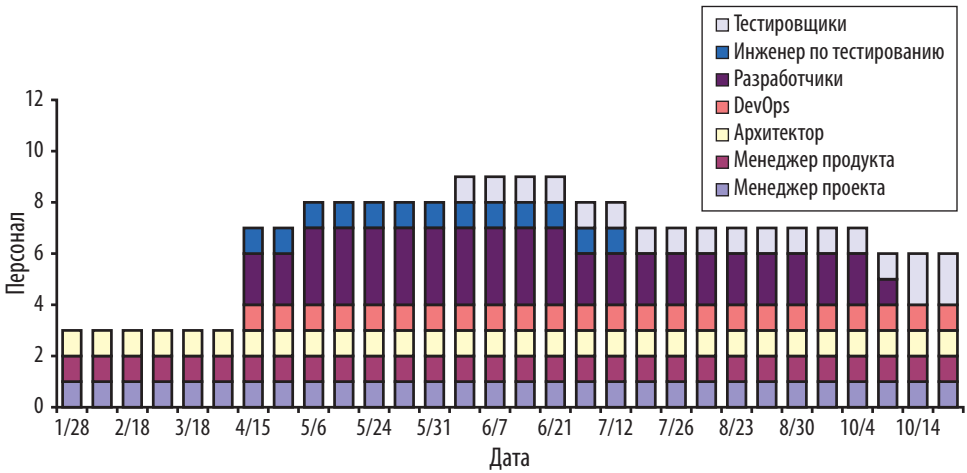


Рис. 11.11. Распределение комплектования с тремя разработчиками и одним инженером по тестированию

Обратите внимание на использование инженера по тестированию с тремя разработчиками. На рис. 11.11 показано лучшее на данный момент распределение комплектования, очень похожее на ожидаемую схему из главы 7 (рис. 7.8).

На рис. 11.12 показана пологая S-образная кривая планируемой осваиваемой ценности. Пологая S-образная кривая получается уж *слишком* пологой (позднее вы увидите, что имеется в виду).

На рис. 11.13 изображена соответствующая диаграмма сети, использующая схему цветового кодирования абсолютной критичности, описанную в главе 8. В примере проекта верхний предел красных активностей составляет 9 дней, а верхний предел для желтых активностей — 26 дней. Идентификаторы активностей выводятся над стрелками черным шрифтом, а значения временного резерва — под линиями такого же цвета, как стрелка. Активности инженера по тестированию — то есть *План тестирования* (активность 4) и *Тестовая оснастка* (активность 5) — имеют очень высокий временной резерв в 65 дней. Обратите внимание на контрольную точку M0, завершающую начальную стадию, и контрольную точку M1 в конце инфраструктуры. На диаграмме также представлен ввод ресурсов между M0 и M1 для построения инфраструктуры (активности 6, 7 и 8).

Без инженера по тестированию (итерация 6)

Следующий эксперимент по сокращению потребления ресурсов — исключить инженера по тестированию, но оставить трех разработчиков. И снова это решение не изменяет ни продолжительности, ни затрат проекта. Третий разработчик просто переходит на активности инженера по тестированию после завершения

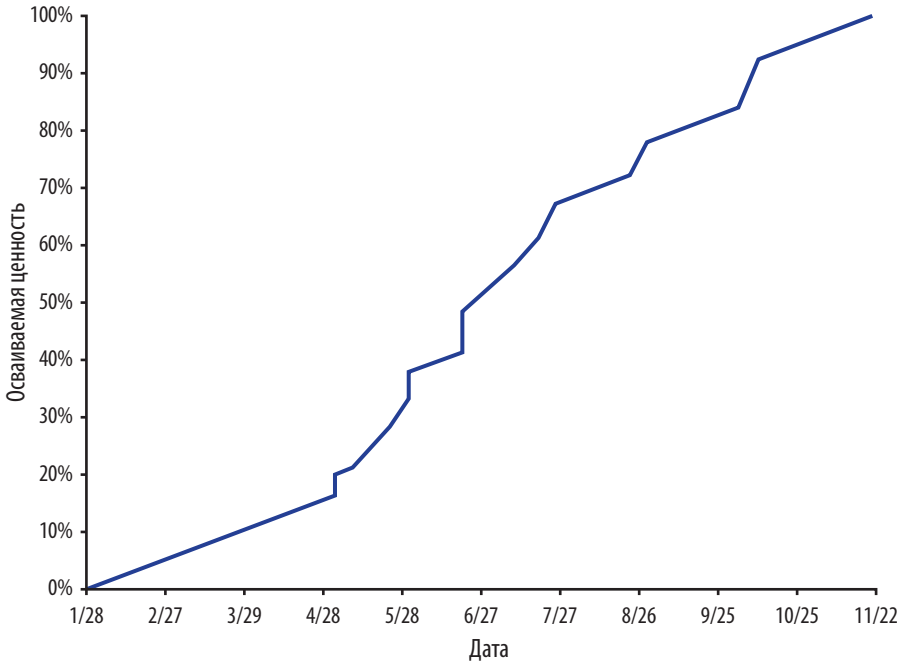


Рис. 11.12. Планируемая осваиваемая ценность с тремя разработчиками и одним инженером по тестированию

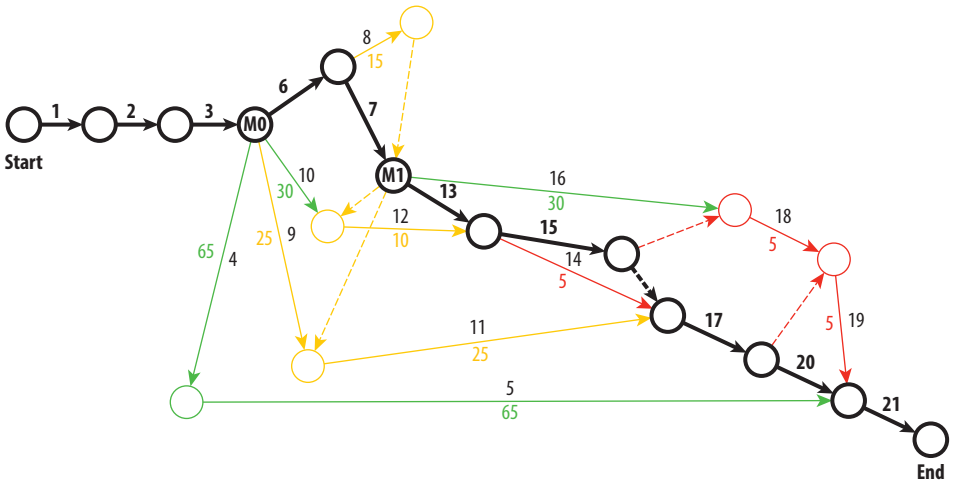


Рис. 11.13. Диаграмма сети с тремя разработчиками и одним инженером по тестированию

других активностей с низким временным резервом, уже назначенных ему. Проблема в том, что перенесение активностей *План тестирования* и *Тестовая оснастка* на более позднее время потребляет 77% их временного резерва (с 65 до 15 дней). Это очень рискованно, потому что падение временного резерва до 100% приведет к задержке проекта.

> **ВНИМАНИЕ** В каждом программном проекте должен присутствовать инженер по тестированию. Инженеры по тестированию настолько важны для успеха проекта, что лучше отказаться от разработчиков, чем отказываться от профессиональных услуг инженеров по тестированию.

Переход на субкритический уровень (итерация 7)

В главе 9 объяснялось, почему так важно представить последствия субкритического комплектования ответственным за принятие решений. Очень часто эти люди не знают о непрактичности урезания ресурсов для предположительного сокращения затрат. В нашем примере проекта для перехода на субкритический уровень количество разработчиков будет ограничено всего двумя, а инженер по тестированию будет исключен. К моменту запуска активностей на критическом пути некоторые поддерживающие некритические активности еще не готовы, поэтому они создают препятствия на старом критическом пути. Ограничивающим фактором становится не продолжительность критического пути, а доступность двух разработчиков. Соответственно, старая сеть (и конкретно старый критический путь) уже неактуальна. Необходимо перерисовать диаграмму сети так, чтобы в ней отражалась зависимость от двух разработчиков.

Вспомните, о чем говорилось в главе 7: зависимости ресурсов являются зависимостями, а сеть проекта представляет собой сеть зависимостей, а не простую сеть активностей. Таким образом, зависимость от ресурсов должна быть добавлена в проект. На самом деле при построении сети существует определенная гибкость: если естественная зависимость между активностями удовлетворяется, фактический порядок активностей можно изменять. Чтобы создать новую сеть, вы назначаете как обычно два ресурса на основании временного резерва. Каждый разработчик после завершения текущей активности берется за следующую доступную активность с наименьшим временным резервом. В то же время вы добавляете зависимость между следующей и текущей активностью разработчика, чтобы отразить зависимость от разработчика. На рис. 11.14 представлена субкритическая диаграмма сети для примера проекта.

С учетом того, что только два разработчика выполняют большую часть работы, субкритическая диаграмма сети выглядит как две длинные цепочки. Одна цепочка активностей содержит длинный критический путь; в другой цепочке второй разработчик оказывает поддержку. Такой длинный критический путь повышает риск проекта, потому что проект теперь содержит больше критиче-

ских активностей. Обычно субкритические проекты всегда являются проектами с высоким риском.

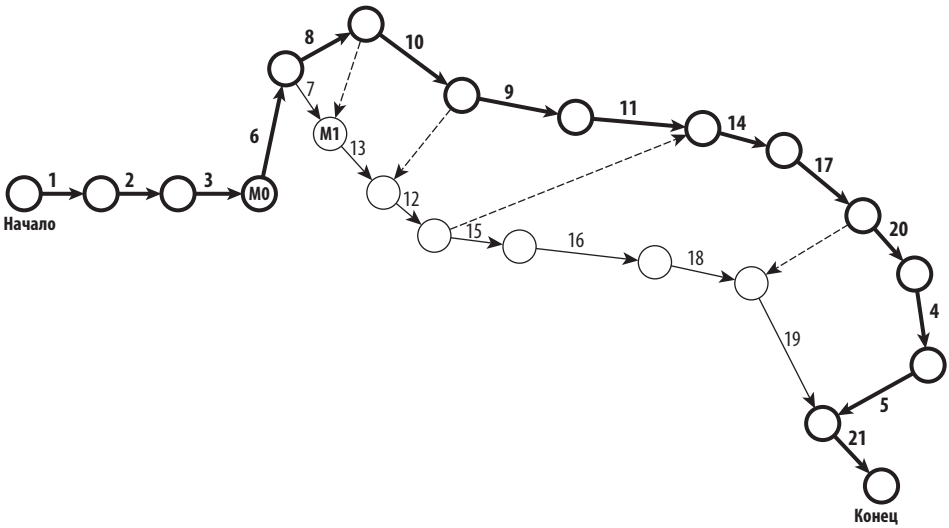


Рис. 11.14. Диаграмма сети субкритического решения

В крайнем случае всего с одним разработчиком все активности проекта являются критическими, диаграмма сети выглядит как одна длинная цепочка, а риск равен 1,0. Продолжительность проекта равна сумме всех активностей, но из-за максимального риска даже такая продолжительность с большой вероятностью будет превышена.

Затраты и продолжительность субкритического проекта

По сравнению с решением с ограниченными ресурсами, в котором задействованы три разработчика и один инженер по тестированию, продолжительность проекта увеличивается на 35% — до 13,4 месяца — вследствие последовательной структуры активностей. Несмотря на сокращение размера команды, общие затраты проекта увеличиваются на 25% до 77,6 человеко-месяца из-за увеличения продолжительности и нарастания косвенных затрат. Результат с очевидностью демонстрирует суть: субкритическое комплектование не дает никакой экономии затрат.

Планируемая осваиваемая ценность

На рис. 11.15 показан график планируемой осваиваемой ценности для субкритического варианта. Как видите, предполагаемая пологая S-образная кривая представляет собой почти прямую линию.

СПЕЦИАЛИЗИРОВАННЫЕ РЕСУРСЫ И ВРЕМЕННОЙ РЕЗЕРВ

На нескольких предыдущих итерациях доступ к специализированным ресурсам (таким, как архитектор базы данных или инженер по тестированию) последовательно ограничивался без заметных последствий для сроков или затрат. Некоторые руководители инстинктивно ожидают такого поведения. Когда архитекторы или другие руководящие специалисты требуют доступа к экспертным ресурсам, руководство часто отклоняет их запросы на том основании, что дополнительные затраты не ускорят реализацию проекта. В этом отношении они правы. Тем не менее ограничение доступа к специализированным ресурсам сокращает временной резерв активностей, назначенных этим экспертам, а это приводит к значительному риску проекта. Многие руководители полностью упускают это следствие. Проекты с высокими ограничениями, в которых задействованы только разработчики, всегда сопряжены с высоким риском. Кроме того, разработчики должны быть (и обычно являются) экспертами в предметной области. Предположения о том, что разработчики являются «мастерами на все руки», нереалистичны, а их результаты часто разочаровывают.

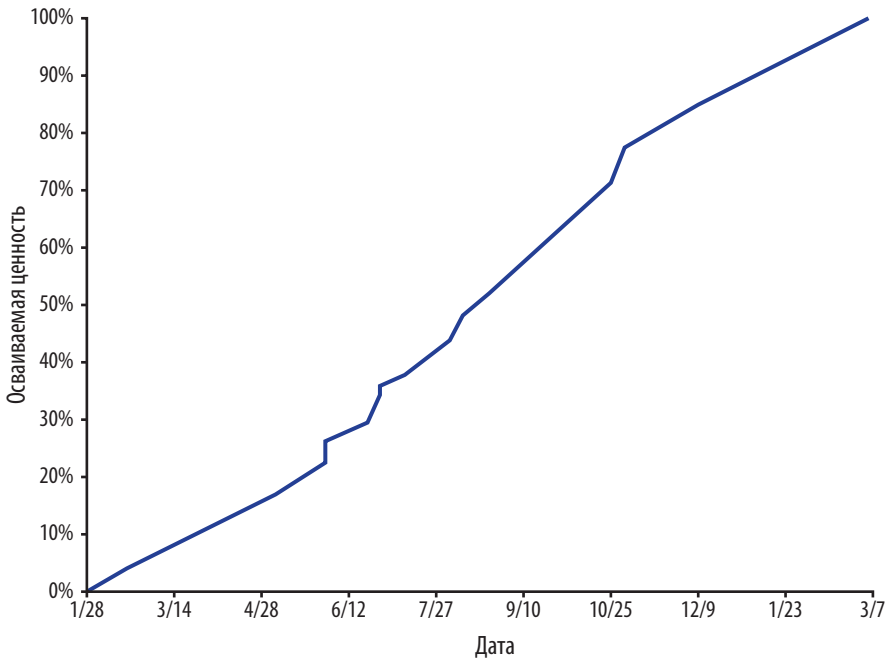


Рис. 11.15. Субкритическая планируемая осваиваемая ценность

В крайнем случае, когда всю работу выполняет один разработчик, планируемая осваиваемая ценность представляет собой прямую линию. В общем случае отсутствие изгиба на диаграмме планируемой осваиваемой ценности — характерный признак субкритического проекта. Даже невыразительная пологая S-образная кривая на рис. 11.12 указывает, что проект близок к субкритическому состоянию.

Выбор нормального решения

Поиск нормального решения состоял из нескольких попыток, использующих разные комбинации ресурсов и вариантов сети. Из всех этих решений до настоящего момента лучшим была итерация 5 (с тремя разработчиками и одним инженером по тестированию) по следующим причинам:

- Такое решение соответствует определению нормального решения: в нем задействован наименьший уровень ресурсов, который позволял проекту беспрепятственно продвигаться вперед по критическому пути.
- Такое решение обходит ограничения доступности экспертов (например, архитектор базы данных) без ущерба доступности ключевого ресурса — инженера по тестированию.
- Такое решение не предполагает, что все разработчики начнут работать одновременно.
- Как диаграмма распределения комплектования, так и диаграмма планируемой осваиваемой ценности демонстрируют приемлемое поведение.

Начальная стадия этого решения охватывала, как и ожидалось, 25% продолжительности проекта, и проект имеет приемлемую эффективность 23%. Вспомните, о чем говорилось в главе 7: для большинства проектов показатель эффективности не должен превышать 25%. В оставшейся части главы итерация 5 используется в качестве нормального решения и базовой линии для последующих итераций. В табл. 11.3 приведена сводка различных метрик проекта для нормального решения.

Таблица 11.3. Метрики проекта для нормального решения

Метрика проекта	Значение
Общие затраты (человеко-месяцы)	61,1
Прямые затраты (человеко-месяцы)	21,8
Продолжительность (месяцы)	9,9
Среднее комплектование	6,1

Таблица 11.3 (*окончание*)

Метрика проекта	Значение
Пиковое комплектование	9
Средняя численность разработчиков	2,3
Эффективность	23%
Начальная стадия	23%

Уплотнение сети

После выбора нормального решения можно попытаться уплотнить проект и понять, насколько эффективно работают некоторые приемы сжатия. Единственно правильного способа уплотнения проектов не существует. Вам неизбежно приходится делать предположения относительно доступности, сложности и затрат. В главе 9 рассматриваются различные методы уплотнения. В общем случае оптимальная стратегия заключается в том, чтобы начать с рассмотрения простых способов уплотнения проекта. В демонстрационных целях в этой главе будет описано уплотнение проекта несколькими способами. Ваша конкретная ситуация будет иной. Возможно, вы выберете некоторые из приемов и идей, описанных здесь, тщательно взвешивая последствия каждого уплотненного решения.

Уплотнение с использованием лучших ресурсов

Простейший способ уплотнения любого проекта — использование лучших ресурсов. Он не требует изменения ни в сети проекта, ни в активностях. Хотя эта форма уплотнения считается простейшей, она может оказаться не самой легкой из-за возможных проблем с доступностью ресурсов (подробнее см. главу 14). Ваша цель — оценить, как проект будет реагировать на уплотнение с использованием лучших ресурсов; понять, стоит ли этим вообще заниматься, и если стоит, то как именно.

Уплотнение с использованием опытного разработчика (итерация 8)

Предположим, у вас имеется опытный разработчик, который может выполнять работу по программированию на 30% быстрее уже имеющихся разработчиков. Вероятно, такой опытный разработчик обойдется намного дороже 30% сверх затрат на обычного разработчика. В нашем проекте предполагается, что опытный разработчик будет стоить на 80% дороже обычного.

В идеале такой ресурс должен назначаться только на критическом пути, но это возможно не всегда (вспомните обсуждение непрерывности задач из главы 7). Базовое нормальное решение назначает двух разработчиков на критический путь, и ваша цель — заменить одного из них самым эффективным ресурсом. Чтобы понять, кого именно следует заменить, необходимо учитывать как количество активностей, так и количество дней, проведенных каждым человеком на критическом пути.

В табл. 11.4 приведены данные двух разработчиков в нормальном решении, количество критических и некритических активностей для каждого, а также общая продолжительность пребывания на критическом пути и за его пределами. Очевидно, *Разработчика 2* лучше заменить опытным разработчиком.

Таблица 11.4. Разработчики, критические активности и продолжительность

Ресурс	Некритические активности	Некритические активности (дни)	Критические активности	Критическая продолжительность (дни)
Разработчик 1	4	8,5	2	35
Разработчик 2	1	5	4	95

Затем необходимо вернуться к табл. 11.1 (оценки продолжительности для всех активностей), выявить активности, за которые отвечает *Разработчик 2*, и снизить их продолжительность на 30% (ожидаемый прирост производительности с лучшим ресурсом) с использованием 5-дневного разрешения. С новыми продолжительностями активностей повторите анализ продолжительности проекта и затрат, учитывая дополнительные 80% затрат на *Разработчика 2*. На рис. 11.16 изображен критический путь на диаграмме сети до и после уплотнения с опытным разработчиком.

Продолжительность нового проекта составляет 9,5 месяца — всего на 4% меньше продолжительности исходного нормального решения. Различия настолько малы, потому что появился новый критический путь, который сдерживает проект. Такое крошечное сокращение продолжительности — довольно распространенный результат. Даже если одиночный ресурс значительно более производителен, чем другие участники команды, и даже если все активности, за которыми закреплен этот высокопроизводительный ресурс, реализуются намного быстрее, продолжительности активностей, над которыми работают рядовые участники команды, от него не зависят, и эти активности просто сдерживают уплотнение.

Что касается затрат, у уплотненного проекта они не изменяются, несмотря на появление высокоэффективного ресурса, который обходится на 80% дороже. Этот факт тоже является ожидаемым из-за косвенных затрат. Обычно про-

граммные проекты обладают высокими косвенными затратами. Даже незначительное снижение продолжительности приблизительно компенсирует дополнительные затраты на уплотнение, по крайней мере при начальных попытках уплотнения, потому что минимум кривой общих затрат располагается слева от нормального решения (см. рис. 9.10).

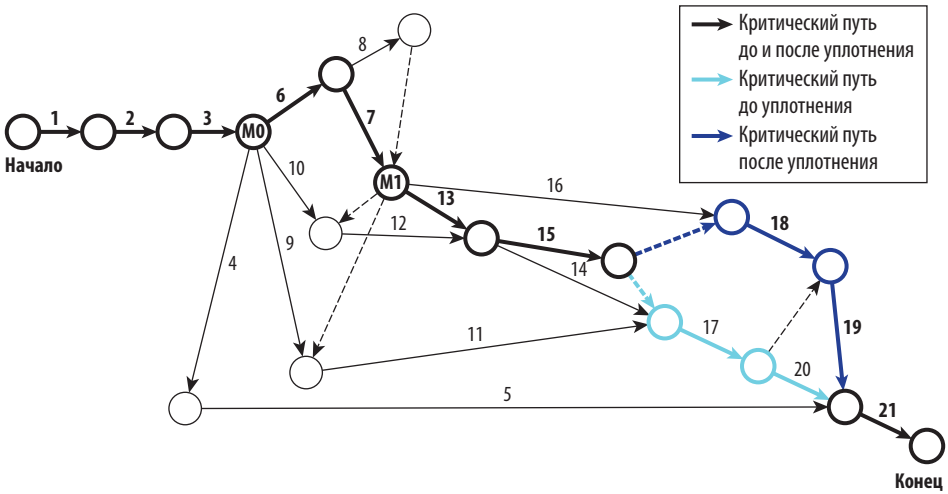


Рис. 11.16. Новый критический путь с одним опытным разработчиком

Уплотнение со вторым опытным разработчиком (итерация 9)

Можно попробовать провести уплотнение за счет использования нескольких высокоэффективных ресурсов. В нашем примере есть смысл попытаться заменить *Разработчика 1* вторым опытным разработчиком, потому что третий опытный разработчик может быть назначен только за пределами критического пути. Со вторым высокоэффективным ресурсом уплотнение имеет более заметный эффект: срок сокращается еще на 11% до 8,5 месяца, а общие затраты сокращаются на 3% — до 59,3 человеко-месяца.

Параллелизм в работе

Часто единственным содержательным способом ускорения работы над проектами становится введение параллелизма в работе. Существует несколько способов параллельной работы в программном проекте, причем одни из них сложнее других. Параллельная работа повышает сложность проекта, поэтому сначала стоит рассмотреть более простые и доступные методы.

Лежащее на поверхности

Лучшими кандидатами для параллельной работы в большинстве хорошо спроектированных систем становятся результаты проектирования инфраструктуры и *Клиента*, потому что они не зависят от бизнес-логики. Ранее эта независимость была задействована в итерации 2, в которой инфраструктура была перемещена в самое начало, чтобы работа над ней запускалась немедленно после анализа SDP. Чтобы обеспечить возможность параллельной работы над *Клиентами*, мы разбиваем *Клиентов* на раздельные активности проектирования и разработки. К числу активностей проектирования, связанных с *Клиентами*, обычно относится UX-проектирование, UI-проектирование, проектирование API и SDK (для взаимодействий с внешними системами). Расщепление *Клиентов* также повышает степень изоляции проектировочных решений *Клиентов* от backend-системы, потому что *Клиенты* должны обеспечивать хороший опыт взаимодействия потребителям сервисов, а не просто отражать нижележащую систему. Теперь разработка инфраструктуры и активности проектирования *Клиентов* может выполняться параллельно в начальной стадии.

Впрочем, у этого подхода есть два недостатка. Меньшим недостатком является повышение начального темпа работ, который возрастает просто из-за того, что с самого начала проекта в нем задействованы как разработчики, так и основная команда. Более серьезный недостаток заключается в том, что начало работы до того, как организация приняла обязательства по проекту, часто заставляет организацию принять решение о продолжении работы, даже если в конкретной ситуации разумнее отменить проект. Человеческая природа заставляет нас игнорировать невозвратные издержки и формирует эффект привязки[1] к красивым прототипам пользовательского интерфейса.

Рекомендую переместить инфраструктуру и проектирование *Клиентов* в начальную стадию только в том случае, если проект гарантированно будет продолжен, а анализ SDP проводится исключительно для выбора варианта (и принятия формальных обязательств по проекту). Риск искажения решения SDP можно частично компенсировать, переместив только разработку инфраструктуры параллельно с начальной стадией, чтобы проектирование *Клиентов* начиналось после анализа SDP. Наконец, убедитесь в том, что активности проектирования *Клиентов* не будут превратно интерпретированы как значительный прогресс в реализации проекта теми, кто приравнивает артефакты пользовательского интерфейса к прогрессу. Работу в начальной стадии следует объединить с отслеживанием статуса проекта (см. приложение А), чтобы ответственные за принятие решений правильно интерпретировали состояние проекта.

---

[1] https://ru.wikipedia.org/wiki/Эффект_привязки

КАК ИЗБЕЖАТЬ ЛОВУШКИ

Перемещение разработки инфраструктуры и проектирования Клиентов в начальную стадию проекта обладает многочисленными преимуществами помимо уплотнения проекта. В главе 7 рассматривается классическая ошибка: ситуация, в которой организация подталкивает руководителей к неверным решениям, комплектуя проект и распределяя функции между разработчиками сразу же после запуска начальной стадии. Так как руководство делает это просто для того, чтобы избежать пустых офисов и бездействия, поручение разработки инфраструктуры и проектирования Клиентов разработчикам обеспечит их занятость, а у основной команды будет время на проектирование системы и проекта. Если анализ SDP (который завершает начальную стадию) закроет проект до того, как будет завершена инфраструктура или проектирование Клиентов, вы просто завершаете эти активности и списываете затраты по ним.

Добавление и расщепление активностей

В других местах проекта выявление дополнительных возможностей для параллельной работы создает больше проблем. Вам придется проявить творческий подход и поискать способы устранения зависимостей между активностями, напрямую связанными с программированием. Это почти всегда требует вложений в дополнительные активности, обеспечивающие параллельную работу, — эмуляторы, имитаторы и интеграционные активности. Вам также придется расщеплять активности и извлекать из них новые активности для подробного проектирования контрактов, интерфейсов, сообщений или проектирования зависимых сервисов. Эти активности проектирования выполняются параллельно с другими активностями.

Для параллельной работы такого рода не существует готовых формул. Она может применяться для нескольких ключевых активностей или для большинства активностей. Дополнительные активности могут выполняться в начале проекта или в процессе работы. Очень быстро вы поймете, что устранить все зависимости между активностями, связанными с программированием, практически невозможно, потому что когда все пути приближаются к критическим, дальнейшие попытки уплотнения перестают давать результаты. Вы будете двигаться по кривой прямых затрат проекта, которая вблизи точки минимальной продолжительности (см. рис. 9.3) характеризуется крутым подъемом, то есть все большими затратами при все меньшем сокращении сроков.

Инфраструктура и проектирование Клиентов в начале (итерация 10)

Возвращаясь к нашему примеру, следующая итерация уплотнения проекта перемещает инфраструктуру для параллельного выполнения с начальной

стадией. Также активности *Клиентов* расщепляются на изначальную работу по проектированию (требования, план тестирования, UI-проектирование) и фактическую разработку *Клиентов*, причем проектирование *Клиентов* переносится в начальную стадию. В нашем примере можно считать, что активности проектирования *Клиентов* не зависят от инфраструктуры и уникальны для каждого *Клиента*.

В табл. 11.5 приведен переработанный набор активностей, их продолжительностей и их зависимостей для данной итерации уплотнения.

Обратите внимание: *Журнал* (активность 6) и, как следствие, остальные активности инфраструктуры вместе с новыми активностями проектирования *Клиентов* (активности 24 и 25) могут стартовать в начале проекта. Также обратите внимание на то, что фактические активности по разработке *Клиентов* (активности 19 и 20) стали короче и теперь зависят от завершения соответствующих активностей проектирования *Клиентов*.

Таблица 11.5. Активности с перемещением инфраструктуры и проектирования Клиентов в начало

ID	Активность	Продолжительность (дни)	Зависит от
1	Требования	15	
2	Архитектура	20	1
3	Планирование проекта	20	2
4	План тестирования	30	22
5	Тестовая оснастка	35	4
6	Журнал	10	
7	Безопасность	15	6
8	Публикация/подписка	5	6
9	Ресурс A	20	22
10	Ресурс B	15	22
11	Доступ к ресурсу A	10	9,23
12	Доступ к ресурсу B	5	10,23
13	Доступ к ресурсу C	10	22,23
14	Ядро A	10	22,23
15	Ядро B	20	12,23
16	Ядро C	10	22,23

Таблица 11.5 (*окончание*)

ID	Активность	Продолжительность (дни)	Зависит от
17	Менеджер A	15	14,15,11
18	Менеджер B	20	15,16
19	Клиентское приложение 1	15	17,18,24
20	Клиентское приложение 2	20	17,25
21	Тестирование системы	30	5,19,20
22	M0	0	3
23	M1	0	7,8
24	Проектирование клиентского приложения 1	10	
25	Проектирование клиентского приложения 2	15	

Уплотнение проекта посредством перемещения инфраструктуры и активностей проектирования Клиентов в начальную стадию создает ряд потенциальных проблем. Первая проблема — это затраты. Продолжительность начальной стадии теперь превышает продолжительность инфраструктуры и активностей проектирования *Клиентов*, даже при последовательном выполнении с теми же ресурсами. Таким образом, начинать эту работу одновременно с начальной стадией было бы неэффективно, потому что разработчики будут бездействовать ближе к концу. Эффективнее будет отложить начало работ по инфраструктуре и проектированию *Клиентов* до того момента, как они станут критическими. Это повысит риск по проекту, но сократит затраты при одновременном уплотнении проекта.

В этой итерации затраты можно дополнительно сократить, используя тех же двух разработчиков для начальной разработки инфраструктуры; после завершения инфраструктуры они переходят к активностям проектирования *Клиентов*. Так как зависимости от ресурсов являются зависимостями, мы определяем активности проектирования *Клиентов* как зависимые от завершения инфраструктуры (M1). Чтобы максимизировать уплотнение, два разработчика, используемые в начальной стадии, переходят к другим активностям проектов (*Ресурсы*) сразу же после анализа SDP (M0). В нашем конкретном случае для правильного вычисления временных резервов мы также делаем анализ SDP зависимым от завершения активностей проектирования *Клиентов*. Тем самым исключается зависимость активностей проектирования *Клиентов* от самих *Клиентов*, а *Клиенты* наследуют зависимость от анализа SDP. Еще раз подчеркну, что мы можем позволить себе переопределить зависимости сети в данном случае только из-за того, что начальная стадия по продолжительности превышает сумму ин-

фраструктуры и активностей проектирования *Клиентов*. В табл. 11.6 приведены пересмотренные зависимости сети (изменения выделены жирным).

Таблица 11.6. Пересмотренные зависимости с перемещением инфраструктуры и проектирования Клиентов в начало

ID	Активность	Продолжительность (дни)	Зависит от
1	Требования	15	
...	...	...	...
19	Клиентское приложение 1	15	17,18,**24**
20	Клиентское приложение 2	20	17,**25**
21	Тестирование системы	30	5,19,20
22	M0	0	3,**24,25**
23	M1	0	7,8
24	Проектирование клиентского приложения 1	10	**23**
25	Проектирование клиентского приложения 2	15	**23**

Другая проблема с расщеплением активностей — повышение сложности *Клиентов* в целом. Эту сложность можно компенсировать, назначив активности проектирования *Клиентов* и их разработку тем же двум разработчикам из предыдущей итерации уплотнения. Таким образом, эффект уплотнения объединяется с эффектом высокоэффективных ресурсов. Но поскольку *Клиенты* и проект становятся более сложными и требовательными, этот факт можно компенсировать предположением о том, что 30-процентное сокращение времени построения *Клиентов* исчезает (хотя разработчики все равно обходятся на 80% дороже). Эти компенсации уже отражены в оценках продолжительности активностей 19 и 20 в табл. 11.5 и 11.6.

Результатом этой итерации уплотнения становится возрастание затрат на 6% по сравнению с предыдущим решением — до 62,6 человеко-месяца и сокращение продолжительности на 8% — до 7,8 месяца. На рис. 11.17 изображена полученная диаграмма сети.

> **ПРИМЕЧАНИЕ** На рис. 11.17 изображена длинная цепочка активностей [10, 12, 15, 18, 19], которая имеет всего от 5 до 10 дней временного резерва. С учетом длины проекта длинные цепочки, имеющие всего 5 (и даже 10) дней временного резерва, должны рассматриваться как критические пути при вычислении риска.

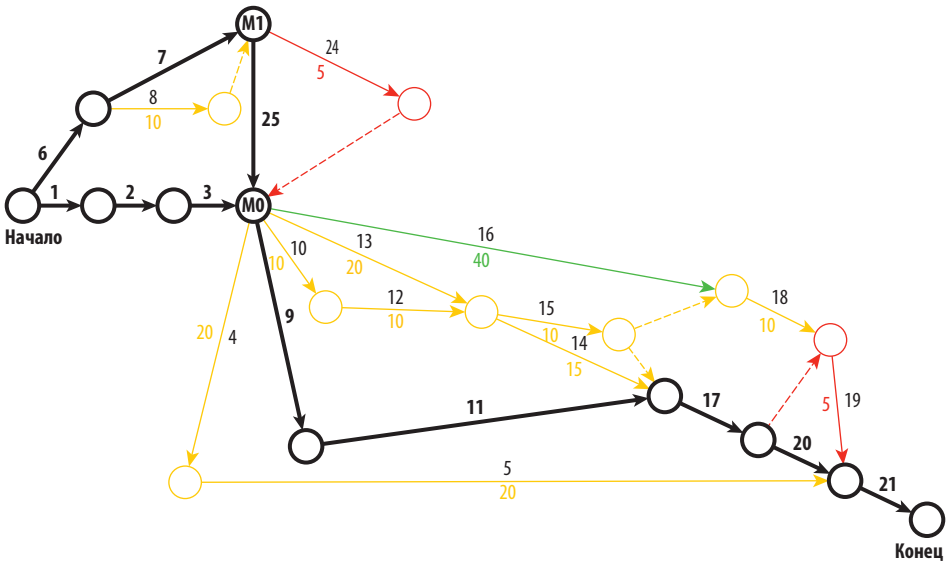


Рис. 11.17. Диаграмма сети с перемещением инфраструктуры и проектирования Клиентов в начало

Сжатие с имитаторами (итерация 11)

Рассматривая рис. 11.17, мы видим, что наряду с критическим путем появился околокритический путь (активности 10, 12, 15, 18 и 19). Это означает, что любое дальнейшее уплотнение потребует уплотнения обоих путей в приблизительно одинаковой степени. Уплотнение всего одного из них не будет иметь особого эффекта, потому что продолжительность проекта в этом случае будет определяться другим путем. В подобных ситуациях лучше искать *корону*, то есть большую активность, которая находится на вершине обоих путей. Уплотнение короны уплотняет оба пути. В примере проекта лучшими кандидатами будут разработка клиентских приложений (активности 19 и 20) и разработка сервисов *Менеджер* (активности 17 и 18). *Клиенты* и *Менеджеры* — относительно большие активности, которые являются коронами для обоих путей. Вы должны попытаться уплотнить *Клиентов* и/или *Менеджеров*.

Чтобы сделать возможным уплотнение *Клиентов*, следует разработать имитаторы (см. главу 9) для сервисов *Менеджеров*, от которых они зависят, и переместить разработку *Клиентов* ближе к началу сети, параллельно с другими активностями. Так как ни один имитатор не является идеальной заменой для реального сервиса, вам также придется выполнить интеграцию между *Клиентами* и *Менеджерами* после того, как *Менеджеры* будут завершены. Фактически это приводит к расщеплению разработки каждого *Клиента* на две активности: первая — активность разработки для имитатора, вторая — активность

интеграции с *Менеджерами*. В результате разработка *Клиентов* может не уплотниться, но общая продолжительность проекта сокращается.

Этот подход можно имитировать посредством разработки имитаторов для *Ядер* и сервисов *Доступ к ресурсам*, от которых зависят *Менеджеры*, что позволит заняться разработкой *Менеджеров* на более ранней стадии проекта. Тем не менее в хорошо спроектированной системе с хорошо спланированным проектом это будет намного сложнее. Хотя моделирование задействованных сервисов потребует написания множества имитаторов и сильно усложнит сеть проекта, настоящей проблемой станет синхронизация. Разработка имитаторов должна происходить более или менее одновременно с разработкой тех сервисов, которые они должны имитировать, так что фактическое уплотнение, которое может быть достигнуто при таком подходе, ограничено. Создание имитаторов для внутренних сервисов следует рассматривать только как крайнюю меру.

В нашем примере лучше всего ограничиться имитацией только *Менеджеров*. Предыдущая итерация уплотнения (перемещение инфраструктуры и проектирования *Клиентов* в начало) дополняется применением имитаторов. При уплотнении этой итерации применяются некоторые новые предположения планирования:

- *Зависимости*. Работа над имитаторами может начинаться после начальной стадии, и они также требуют инфраструктуры. Это достигается зависимостью от M0 (активность 22).
- *Дополнительные разработчики*. При использовании предыдущей итерации уплотнения как начальной точки потребуются два дополнительных разработчика для разработки имитаторов и реализации *Клиентов*.
- *Начальная точка*. Затраты на двух дополнительных разработчиков можно сократить, отложив работу над имитаторами и *Клиентами* до того момента, когда они станут критическими. Тем не менее сети с имитаторами обычно бывают довольно сложными. Чтобы компенсировать эту сложность, желательно начать работу над имитаторами как можно раньше, чтобы проект получил выигрыш от более высокого временного резерва (вместо снижения затрат).

В табл. 11.7 перечислены активности и изменения в зависимостях. Предыдущая итерация используется в качестве базового решения (изменения выделены жирным).

Полученная диаграмма распределения комплектования изображена на рис. 11.18. На диаграмме хорошо виден резкий рост численности разработчиков после начальной стадии и почти постоянный уровень использования ресурсов. В этом решении в среднем задействовано 8,9 человека, при этом пиковое комплектование достигает 11. По сравнению с предыдущей итерацией решение с имитаторами приводит к 9-процентному сокращению продолжительности — до 7,1 месяца, но общие затраты возрастают всего на 1% — до 63,5 человеко-месяца. Незначительное повышение затрат обусловлено сокращением косвенных затрат

при повышении эффективности и ожидаемой результативности команды при параллельной работе.

Таблица 11.7. Активности с имитаторами Менеджеров

ID	Активность	Продолжительность (дни)	Зависит от
1	Требования	15	
...	...	...	...
17	Менеджер A	15	
18	Менеджер B	20	
19	Клиентское приложение 1 — **интеграция**	15	17,18**,28**
20	Клиентское приложение 2 — **интеграция**	20	17**,29**
...	...	...	...
26	**Имитатор Менеджера A**	**15**	**22**
27	**Имитатор Менеджера B**	**20**	**22**
28	Клиентское приложение 1	15	**26,27**
29	Клиентское приложение 2	20	**26**

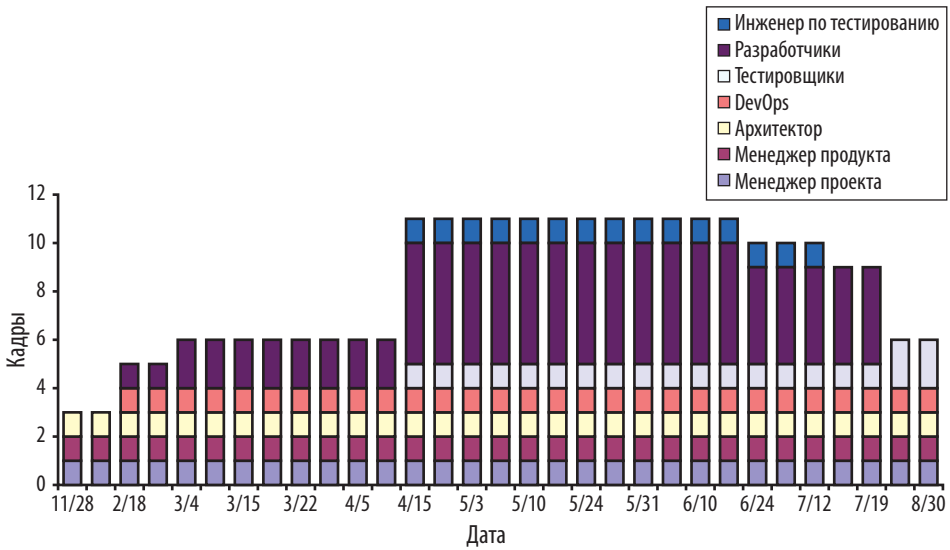


Рис. 11.18. Диаграмма распределения комплектования в решении с имитаторами

На рис. 11.19 изображена диаграмма сети решения с имитаторами. На ней виден высокий временной резерв для имитаторов (активности 26 и 27) и разработки *Клиента* (активности 28 и 29). Также заметим, что практически все остальные сетевые пути являются критическими или околокритическими, и ближе к концу проекта возникает высокое интеграционное давление. Такое решение неустойчиво к непредвиденным факторам, а сложность сети заметно повышает риск исполнения.

Завершение итераций уплотнения

По сравнению с нормальным решением решение с имитаторами сокращает дедлайн на 28%, тогда как затраты возрастают всего на 4%. Из-за высоких косвенных затрат уплотнение практически оправдывает себя. С другой стороны, прямые затраты повышаются на 59% — по процентам вдвое больше сокращения продолжительности. Как упоминалось в главе 9, максимальное ожидаемое уплотнение в программных проектах составляет не более 30%, так что решение с имитаторами обеспечивает такое уплотнение, которого вообще можно ожидать от такого проекта.

Хотя дальнейшее сжатие теоретически возможно (за счет уплотнения *Менеджеров*), на практике это тот максимум, к которому должна стремиться команда планирования проекта. Дальнейшие попытки уплотнения имеют слишком низкую вероятность успеха, и основная команда будет только попусту тратить время на проектирование маловероятных проектов.

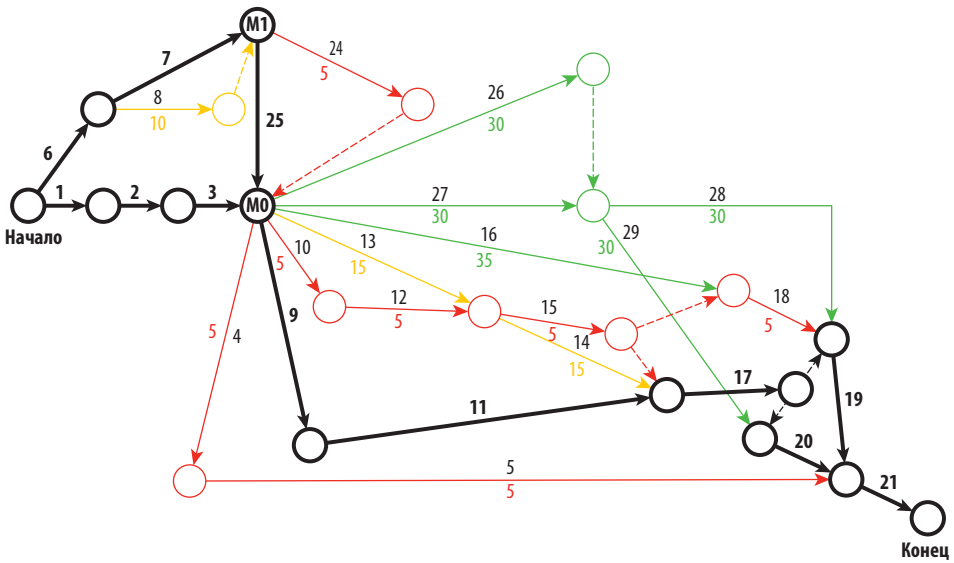


Рис. 11.19. Диаграмма сети решения с имитаторами

Анализ результативности

Важно понимать, как уплотнение влияет на ожидаемую результативность команды по сравнению с нормальным решением. Как объяснялось в главе 7, угол наклона S-образной кривой представляет результативность команды. На рис. 11.20 изображены S-образные кривые планируемой осваиваемой ценности для нормального решения и каждого уплотненного решения в одном масштабе.

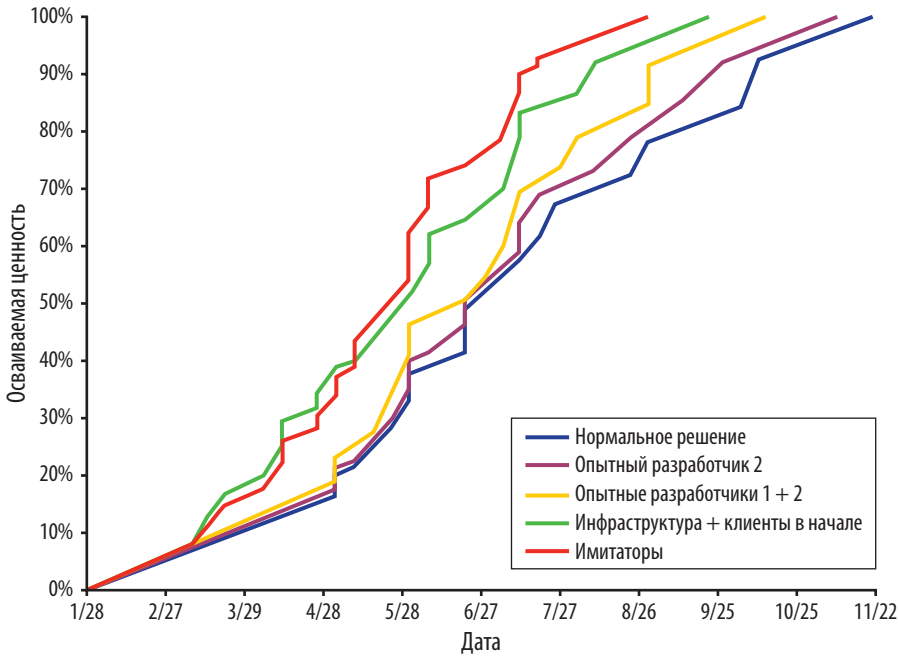


Рис. 11.20. Планируемая осваиваемая ценность разных решений

Как и ожидалось, уплотненные решения имеют более крутую S-образную кривую, потому что они завершаются быстрее. Чтобы получить числовую оценку различий в требуемой результативности, можно заменить каждую кривую соответствующей линией тренда линейной регрессии и проанализировать уравнение линии (рис. 11.21).

Коэффициент составляющей x определяет угол наклона линии, то есть ожидаемую результативность команды. В случае нормального решения команда должна работать на 39 единицах производительности, тогда как решение с имитаторами требует 59 единиц производительности (0,0039 против 0,0059, масштабированные до целых чисел). Конкретная природа этих единиц производительности несущественна. Важны различия между двумя решениями: решение с имитаторами ожидает 51-процентного роста результативности команды

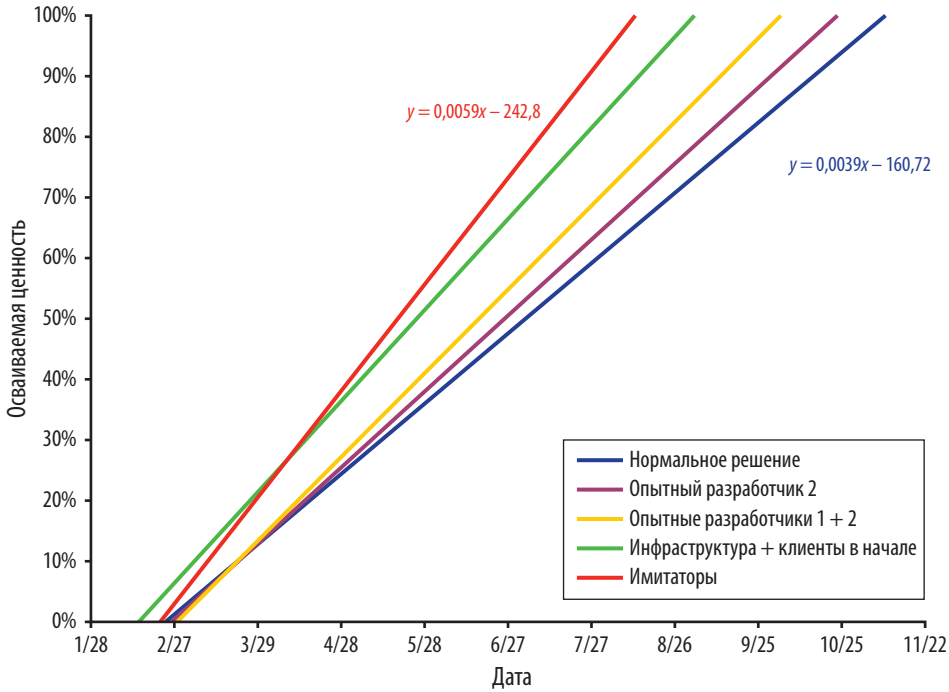


Рис. 11.21. Линии тренда осваиваемой ценности разных решений

(59 – 39 = 20, что составляет 51% от 39). Вряд ли любая команда, даже при увеличении размера, сможет повысить свою результативность в такой степени.

Сравнение отношения среднего комплектования к пиковому для двух решений дает некоторое представление о том, насколько реалистичны различия в результативности (хотя и не является абсолютно надежным правилом). Для решения с имитаторами это отношение составляет 81% по сравнению с 68% для нормального решения; иначе говоря, решение с имитаторами предполагает более интенсивное использование ресурсов. Так как решение с имитаторами также требует большего среднего размера команды (8,9 против 6,1 у нормального решения), а большие команды обычно работают менее эффективно, вероятность достижения 51-процентного роста результативности выглядит сомнительной, особенно при работе над более сложным проектом. Это дополнительно подкрепляет идею о том, что для большинства команд решение с имитаторами устанавливает слишком высокую планку.

Анализ эффективности

Эффективность каждого решения при планировании проекта вычисляется относительно легко и предоставляет много полезной информации. Вспомните,

о чем говорилось в главе 7: числовой показатель эффективности описывает как ожидаемую эффективность команды, так и реалистичность предположений плана относительно ограничений, эластичности комплектования и критичности проекта. На рис. 11.22 изображена диаграмма эффективности решений для проекта из нашего примера.

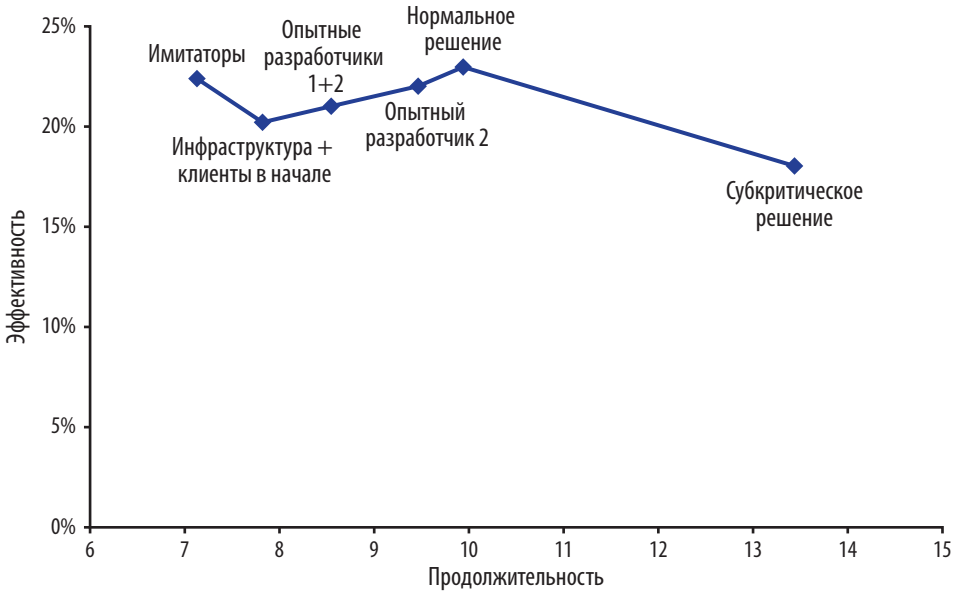


Рис. 11.22. Диаграмма эффективности проекта

На рис. 11.22 пиковая эффективность достигается в нормальном решении, обеспечивающем наименьший уровень использования ресурсов без затрат на уплотнение. При уплотнении проекта эффективность снижается. Хотя решение с имитаторами находится на одном уровне с нормальным решением, я считаю его нереалистичным, потому что проект существенно усложняется, а его реализуемость оказывается под вопросом (как показывает анализ результативности). Субкритическое решение очень плохо в отношении эффективности из-за плохого отношения прямых затрат к косвенным. Короче говоря, нормальное решение является наиболее эффективным.

Кривая «время-затраты»

После проектирования каждого решения и построения его диаграммы распределения комплектования можно вычислить элементы затрат для каждого решения, как видно из табл. 11.8.

Таблица 11.8. Продолжительность, общие затраты и элементы затрат для разных вариантов

Вариант проектирования	Продолжительность (месяцы)	Общие затраты (человеко-месяцы)	Прямые затраты (человеко-месяцы)	Косвенные затраты (человеко-месяцы)
Имитаторы	7,1	63,5	34,8	28,7
Инфраструктура + Клиенты в начале	7,8	62,6	30,4	32,2
Опытные разработчики 1+2	8,5	59,3	26,6	32,7
Опытный разработчик 2	9,5	61,1	24,2	36,9
Нормальное решение	9,9	61,1	21,8	39,2
Субкритическое решение	13,4	77,6	20,9	56,7

С этими показателями затрат можно построить кривые «время-затраты», изображенные на рис. 11.23. Обратите внимание: кривая прямых затрат получается пологой из-за масштабирования диаграммы. Кривая косвенных затрат является почти идеальной прямой.

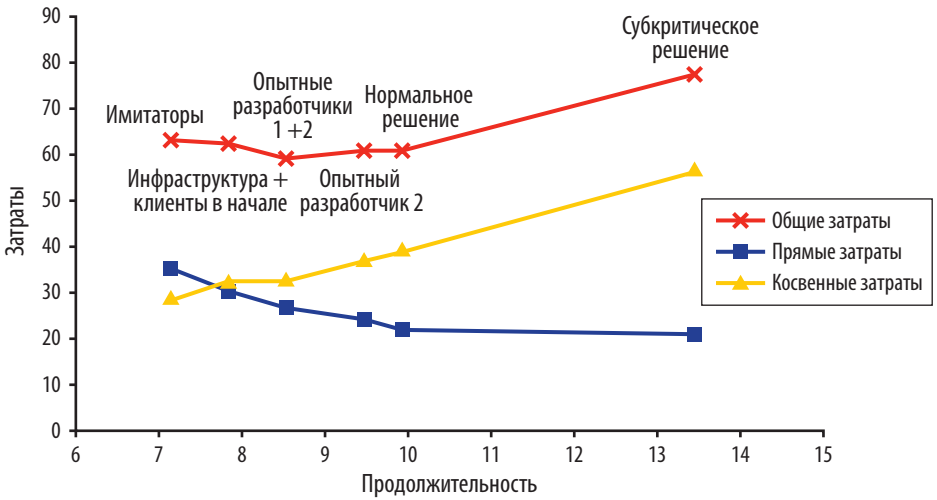


Рис. 11.23. Кривые «время-затраты» проекта

Корреляционные модели «время-затраты»

Кривые «время-затраты» на рис. 11.23 дискретны, и они могут только давать общее представление о поведении кривых за пределами конкретных решений. Тем не менее при наличии дискретных кривых «время-затраты» также возможно найти корреляционные модели для кривых. Корреляционная модель, или линия тренда, — математическая модель, которая строит кривую, оптимальным образом аппроксимирующую дискретные точки данных (такие программы, как Microsoft Excel, позволяют легко выполнять такой анализ). Корреляционные модели позволяют получить значение кривой «время-затраты» в любой точке (а не только в точках известных дискретных решений). Для точек на рис. 11.23 эти модели представляют прямую линию для косвенных затрат и полиномиальную кривую второй степени для прямых и косвенных затрат. На рис. 11.24 корреляционные линии трендов обозначены пунктирными линиями, с указанием формул и значений R^2.

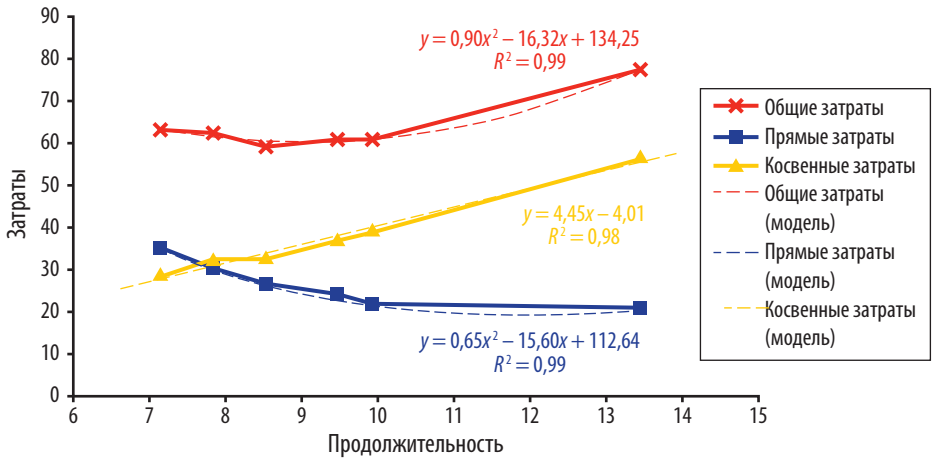


Рис. 11.24. Линии тренда кривых «время-затраты» проекта

Метрика R^2 (также называемая *коэффициентом детерминации*) представляет собой число в диапазоне от 0 до 1, представляющее качество модели. Числа, большие 0,9, указывают на превосходную подгонку модели к дискретным точкам. В этом случае формулы в диапазоне решений планирования проекта очень точно представляют их кривые.

На рис. 11.24 приведены формулы изменения затрат со временем для проекта из нашего примера. Для прямых и косвенных затрат формулы выглядят так:

$$\text{Прямые затраты} = 0{,}65t^2 - 15{,}6t + 112{,}64;$$
$$\text{Косвенные затраты} = 4{,}45t - 4{,}01,$$

где t измеряется в месяцах. Хотя вы также получаете корреляционную модель для общих затрат, эта модель формируется на основе статистических вычислений, поэтому она не является точной суммой прямых и косвенных затрат. Правильная модель общих затрат строится простым суммированием формул прямой и косвенной модели:

$$\begin{aligned}\text{Общие затраты} &= \text{Прямые затраты} + \text{Косвенные затраты} = \\ &= 0{,}65t^2 - 15{,}6t + 112{,}64 + 4{,}45t - 4{,}01 = \\ &= 0{,}65t^2 - 11{,}15t + 108{,}63.\end{aligned}$$

На рис. 11.25 изображена модифицированная корреляционная модель общих затрат вместе с прямой и косвенной моделью.

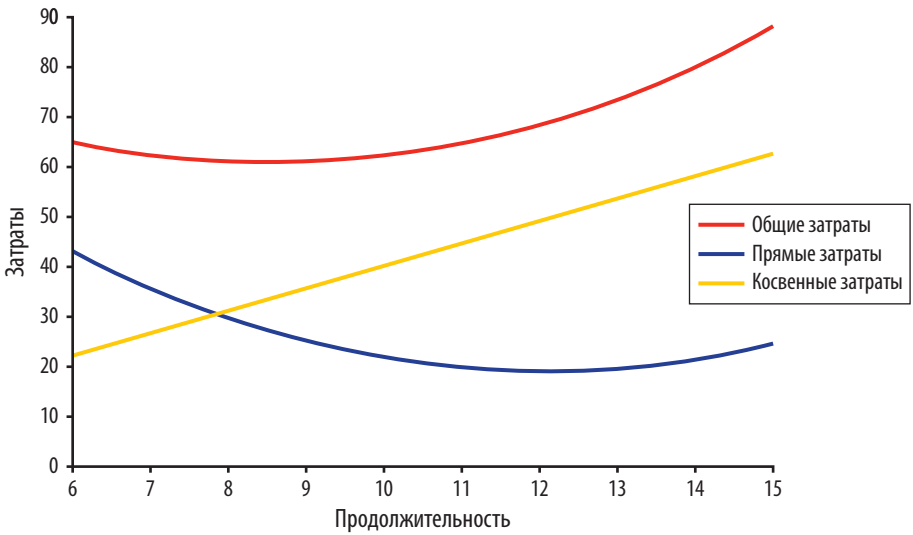


Рис. 11.25. Модели «время-затраты» для проекта

Мертвая зона

В главе 9 приведена концепция мертвой зоны, то есть области под кривой «время-затраты». Любое решение по планированию проекта, которое попадает в эту область, не может быть построено. Наличие модели (или даже дискретной кривой) для общих затрат проекта позволяет наглядно представить мертвую зону проекта, как показано на рис. 11.26.

Определение мертвой зоны позволяет разумно отвечать на быстрые вопросы и избегать принятия обязательств по невозможным проектам. Например,

представьте, что руководство спрашивает вас, возможно ли построить проект за 9 месяцев с 4 людьми. По данным модели общих затрат проекта затраты 9-месячного проекта превышают 60 человеко-месяцев и требуют в среднем 7 участников:

$$\text{Общие затраты} = 0{,}65 \times 9^2 - 11{,}15 \times 9 + 108{,}63 = 61{,}2.$$
$$\text{Среднее комплектование} = 61{,}2/9 = 6{,}8.$$

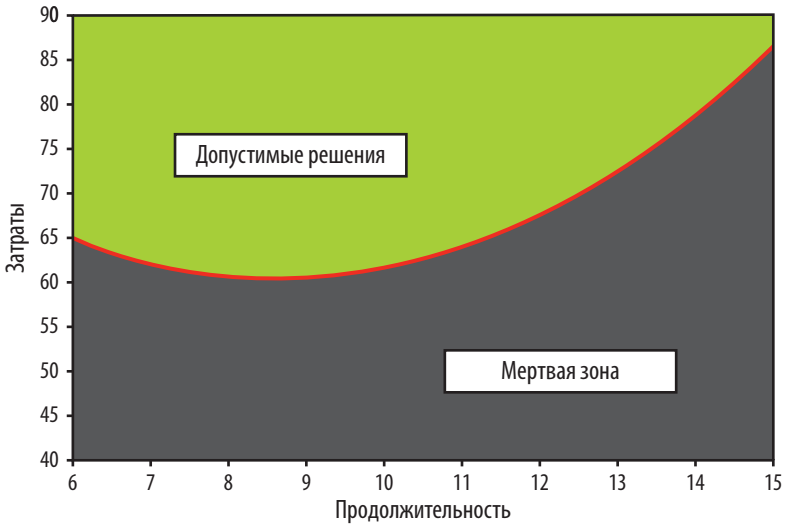


Рис. 11.26. Мертвая зона проекта

Предполагая такое же отношение среднего комплектования с пиковым, как у нормального решения (68%), решение, реализуемое за 9 месяцев, достигает пика в 10 человек. При любой численности, меньшей 10, проект временами выходит на субкритический уровень. Комбинация из 4 участников и 9 месяцев (даже при 100-процентной эффективности использования на 100% времени) дает затраты в 36 человеко-месяцев. Эта конкретная точка в координатах времени/затрат находится так глубоко внутри мертвой зоны, что на рис. 11.26 она даже не видна. Представьте эти результаты руководству и спросите, хотят ли они принимать обязательства на таких условиях.

ПРИМЕЧАНИЕ Достаточно часто архитекторы или руководители проектов интуитивно чувствуют, что какое-то решение руководства неприемлемо, но им не хватает инструментов или данных для того, чтобы это решение опротестовать. Планирование проекта — объективный и бесконфликтный способ представления фактов и обсуждения реальности.

Планирование и риски

Каждое решение по планированию проекта подразумевает некоторый уровень риска. Применяя методы моделирования риска, описанные в главе 10, можно найти численную оценку этих уровней риска для решений (табл. 11.9).

Таблица 11.9. Уровни риска разных решений

Вариант	Продолжительность (месяцы)	Риск возникновения критичности	Риск активности
Имитаторы	7,1	0,81	0,76
Инфраструктура + Клиенты в начале	7,8	0,77	0,81
Опытные разработчики 1 + 2	8,5	0,79	0,80
Опытный разработчик 2	9,5	0,70	0,77
Нормальное решение	9,9	0,73	0,79
Субкритическое решение	13,4	0,79	0,79

На рис. 11.27 показаны уровни рисков вариантов планирования проекта вместе с кривой прямых затрат. Из диаграммы можно сделать как отрицательные, так и положительные выводы относительно риска. Хорошая новость: риск возникновения критичности и риски активностей в этом проекте идут близко друг к другу. Схождение разных моделей по числам всегда является хорошим

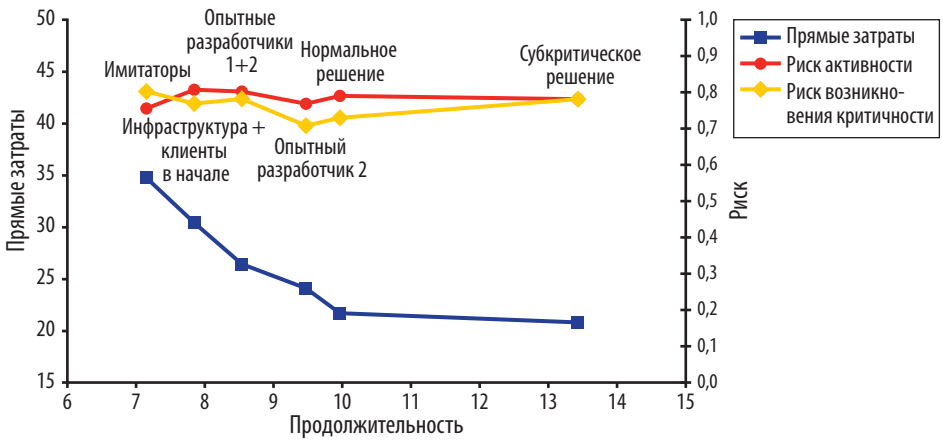


Рис. 11.27. Прямые затраты и риски разных вариантов

признаком — оно придает достоверность значениям. Плохая новость: все рассмотренные решения по планированию проекта являются рискованными решениями; что еще хуже, все они обладают примерно одинаковой ценностью. Это означает, что риск остается высоким и приблизительно одинаковым независимо от решения. Другая проблема заключается в том, что рис. 11.27 содержит субкритическую точку. Субкритических решений определенно следует избегать, и их следует исключить из этой и всех последующих попыток анализа.

В общем случае моделирование не должно базироваться на плохих вариантах планирования. Чтобы решить проблему высокого риска, следует провести разуплотнение проекта.

Разуплотнение риска

Так как в нашем примере проекта все решения имеют достаточно высокий риск, вам следует разуплотнить нормальное решение и внедрить временной резерв в критический путь до тех пор, пока риск не упадет до приемлемого уровня. Разуплотнение проекта является итеративным процессом, потому что вы не знаете заранее, до какой степени следует проводить разуплотнение или как проект отреагирует на разуплотнение.

Первая итерация разуплотняет проект на 3,5 месяца, от 9,9 месяца нормального решения до дальней точки субкритического решения. Результат показывает, как проект реагирует на разуплотнение на всем диапазоне продолжительностей решения. При этом будет получена точка разуплотнения D1 (общая продолжительность проекта 13,4 месяца) с риском возникновения критичности 0,29 и риском активности 0,39. Как объяснялось в главе 10, значение 0,3 должно быть наименьшим уровнем риска для любого проекта, из чего следует, что итерация приводит к чрезмерному разуплотнению проекта.

На следующей итерации проект разуплотняется на 2 месяца по сравнению с нормальной продолжительностью — приблизительно наполовину от величины разуплотнения D1. В результате будет получена точка D2 (общая продолжительность проекта 12 месяцев). Риск возникновения критичности не изменился и остался равным 0,29, потому что эти 2 месяца разуплотнения все еще больше нижнего предела, использованного в этом проекте для зеленых активностей. Риск активности повышается до 0,49.

Аналогичным образом при половинном разуплотнении от D2 вы получаете точку D3 с 1-месячным разуплотнением (общая продолжительность проекта составляет 10,9 месяца), риском возникновения критичности 0,43 и риском активности 0,62. Половина от D3 дает точку D4 с 2-недельным разуплотнением (общая продолжительность проекта 10,4 месяца), риском возникновения критичности 0,45 и риском активности 0,7. На рис. 11.28 показаны кривые рисков после разуплотнения.

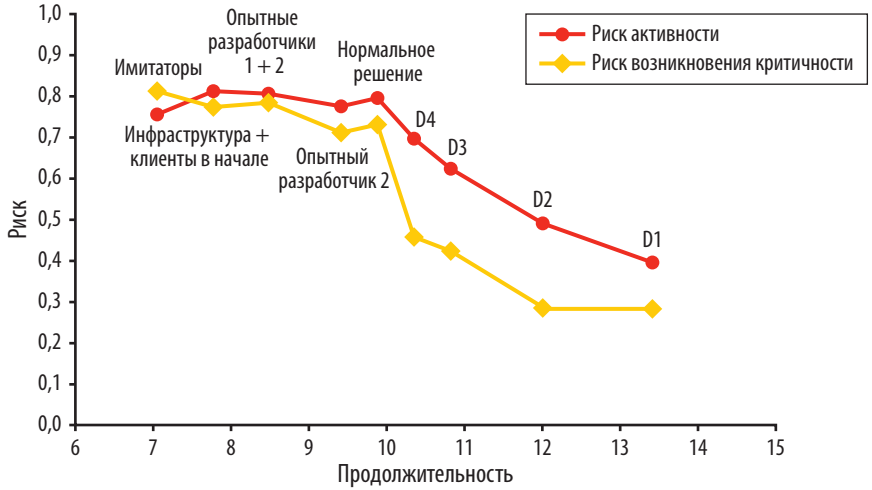


Рис. 11.28. Кривые рисков при разуплотнении

Корректировка выбросов

На рис. 11.28 представлен хорошо видимый разрыв между двумя моделями рисков. Это различие обусловлено ограничением модели риска активности — дело в том, что модель риска активности некорректно вычисляет значения риска в том случае, когда временные резервы в проекте не распределены равномерно (за подробностями обращайтесь к главе 10). В случае разуплотненных решений высокие значения временных резервов для плана тестирования и тестовой оснастки приводят к дополнительному завышению значений рисков активностей. Высокие значения временных резервов отходят от среднего значения всех временных резервов на величину более одного стандартного отклонения, в результате чего они рассматриваются как статистические выбросы.

При вычислении риска активности в точках разуплотнения можно скорректировать входные данные, заменив временной резерв аномальных активностей величиной, полученной суммированием среднего значения всех временных резервов и одного стандартного отклонения всех временных резервов. Электронная таблица позволит легко автоматизировать корректировку всех выбросов. Обычно такая корректировка улучшает корреляцию моделей рисков.

На рис. 11.29 изображена скорректированная кривая риска активности вместе с кривой риска возникновения критичности. Как видите, две модели риска теперь лучше соответствуют друг другу.

Точка перегиба риска

Самый важный аспект на рис. 11.29 — точка перегиба риска возле D4. Даже незначительное дальнейшее разуплотнение проекта к D4 приводит к значитель-

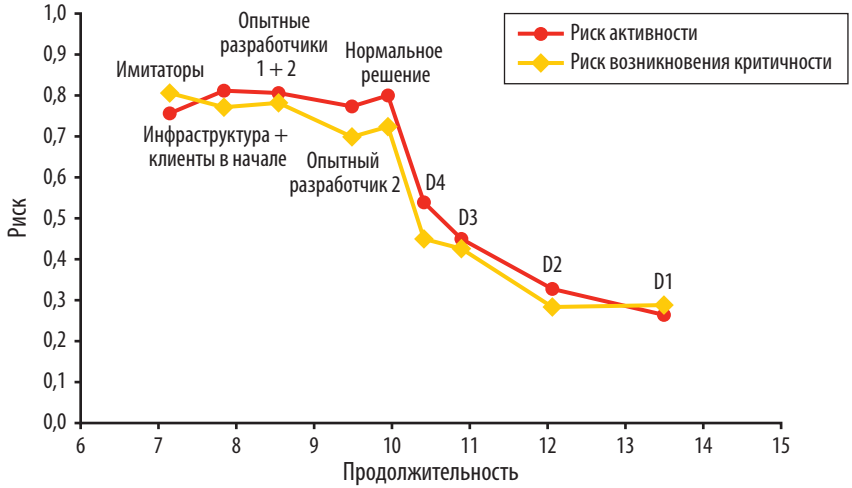


Рис. 11.29. Кривая риска возникновения критичности и скорректированная кривая риска активности при разуплотнении

ному снижению риска. Так как D4 находится непосредственно вблизи от точки перегиба, вам следует действовать более консервативно и проводить разуплотнение в направлении D3, чтобы обойти перегиб в кривых.

Прямые затраты и разуплотнение

Чтобы сравнить разуплотненные решения с другими решениями, необходимо знать соответствующие затраты. Проблема в том, что точки разуплотнения предоставляют только продолжительность и риск. Никакие варианты планирования проекта не дают эти точки — они всего лишь являются значениями риска для сети нормального решения с дополнительным временным резервом. Вам придется экстраполировать как косвенные, так и прямые затраты разуплотненных решений по известным решениям.

В нашем примере косвенные затраты модели представляют собой прямую линию, поэтому косвенные затраты надежно экстраполируются по косвенным затратам другого решения (исключая субкритическое). Например, в результате экстраполяции для D1 будут получены косвенные затраты в размере 51,1 человеко-месяца.

При экстраполяции прямых затрат придется решать проблему с задержками. Дополнительные прямые затраты (помимо нормального решения, которое использовалось для создания разуплотненных решений) происходят как из-за удлинения критического пути, так и из-за увеличения времени бездействия между некритическими активностями. Так как комплектование не является ни полностью динамическим, ни эластичным, возникновение задержки часто

приводит к тому, что людям в других цепочках приходится простаивать в ожидании критических активностей.

В нормальном решении нашего проекта после начальной стадии прямые затраты в основном формируются за счет разработчиков. Также вклад в прямые затраты вносят активности инженера по тестированию и завершающее тестирование системы. Так как инженер по тестированию имеет очень большой временной резерв, можно предполагать, что отставания от графика не будут влиять на инженера по тестированию. Распределение комплектования для нормального решения (на рис. 11.11) показывает, что комплектование достигает пика при трех разработчиках (причем даже этот пик сохраняется ненадолго), а потом сокращается до одного разработчика. Из табл. 11.3 видно, что нормальное решение в среднем использует 2,3 разработчика. Следовательно, можно полагать, что разуплотнение влияет на двух разработчиков. Один из них потребляет временной резерв разуплотнения, а другой в конечном итоге простаивает.

Предпосылки планирования в этом проекте утверждают, что разработчики между активностями учитываются как прямые затраты. Затем, когда в проекте происходит смещение, оно добавляет прямые затраты на двух разработчиков, умноженные на разность продолжительностей нормального решения и точки разуплотнения. В случае самой дальней точки разуплотнения D1 (13,4 месяца) разность в продолжительности относительно нормального решения (9,9 месяца) составляет 3,5 месяца, так что дополнительные прямые затраты составят 7 человеко-месяцев. Так как нормальное решение составляет 21,8 человеко-месяца в части прямых затрат, прямые затраты D1 составят 28,8 человеко-месяца. Также можно добавить другие точки разуплотнения, выполнив аналогичные вычисления. На рис. 11.30 изображена модифицированная кривая прямых затрат с кривыми риска.

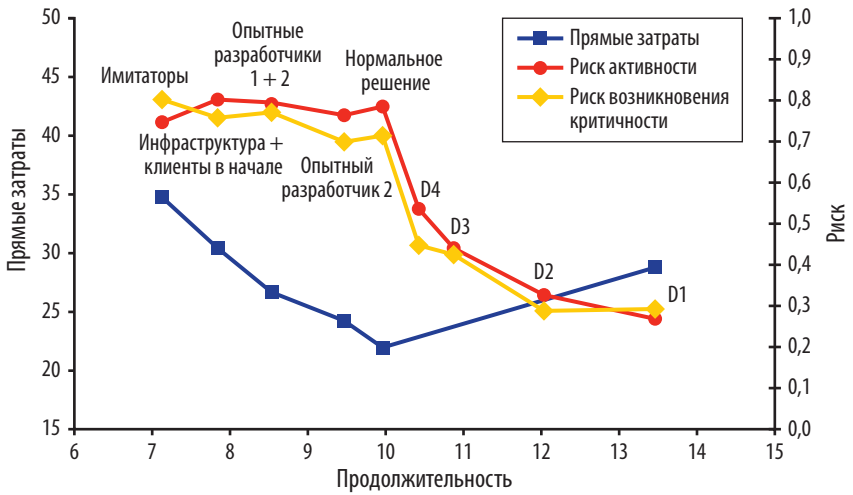


Рис. 11.30. Модифицированная кривая прямых затрат и кривые рисков

Переработка кривой «время-затраты»

С новыми показателями затрат для D1 можно перестроить кривые «время-затраты», исключив точку аномальных данных для субкритического решения. В результате будет получена улучшенная кривая «время-затраты», основанная на возможных решениях. После этого можно переходить к вычислению корреляционных моделей, как и прежде. Процесс дает следующие формулы затрат:

$$\text{Прямые затраты} = 0{,}99t^2 - 21{,}32t + 136{,}57;$$
$$\text{Косвенные затраты} = 3{,}54t + 3{,}59;$$
$$\text{Общие затраты} = 0{,}99t^2 - 17{,}78t + 140{,}16.$$

У этих кривых метрика R^2 равна 0,99, что свидетельствует о превосходной подгонке к точкам данных. На рис. 11.31 изображены новые модели кривых «время-затраты», а также точки минимальных общих затрат и нормального решения.

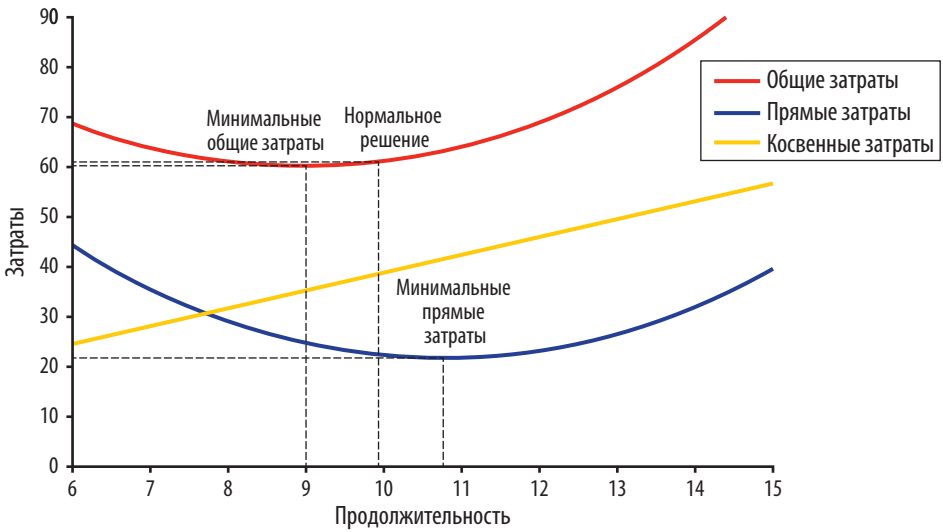


Рис. 11.31. Переработанные модели кривых «время-затраты»

Зная улучшенную формулу общих затрат, вы можете вычислить точку минимальных общих затрат для проекта. Модель общих затрат представляет собой квадратный многочлен следующего вида:

$$y = ax^2 + bx + c.$$

Как известно из курса математического анализа, в точке минимума такого многочлена первая производная равна нулю:

$$y' = 2ax + b = 0;$$

$$x_{\min} = -\frac{b}{2a} = -\frac{-17{,}78}{2 \times 0{,}99} = 9{,}0.$$

Как было показано в главе 9, точка минимума общих затрат всегда смещена влево от нормального решения. Хотя точное решение минимальных общих затрат неизвестно, в главе 9 предполагается, что для большинства проектов поиск этой точки не стоит затраченных усилий. Вместо этого можно для простоты приравнять общие затраты нормального решения к минимальным общим затратам проекта. В этом случае минимальные общие затраты составляют 60,3 человеко-месяца, а общие затраты нормального решения в соответствии с моделью составляют 61,2 человеко-месяца — разность всего 1,5%. Очевидно, упрощающее предположение в данном случае оправданно. Если вашей целью является минимизация общих затрат, то как нормальное решение, так и первое уплотненное решение с одним опытным разработчиком являются приемлемыми вариантами.

Минимальные прямые затраты

Выполнив аналогичные действия с формулой прямых затрат, можно легко вычислить точку во времени с минимальными прямыми затратами 10,8 месяца. В идеале нормальное решение также является точкой минимальных прямых затрат. Тем не менее в нашем примере нормальное решение составляет 9,9 месяца. Такое расхождение отчасти обусловлено различиями между дискретной моделью проекта и непрерывной моделью (см. рис. 11.30, где нормальное решение также является точкой минимума прямых затрат в отличие от рис. 11.31). Более содержательная причина заключается в том, что точка сместилась вследствие переработки кривой «время-затраты» с учетом точек разуплотнения риска. На практике нормальное решение часто немного смещается от точки минимальных прямых затрат из-за воздействия ограничений. В оставшейся части главы продолжительность 10,8 месяца используется как точка минимума прямых затрат.

Моделирование риска

Теперь можно построить модели линий тренда для дискретных моделей риска, как показано на рис. 11.32. Две линии тренда на этой диаграмме очень похожи. В оставшейся части главы будет использоваться линия тренда риска активности из-за ее большей консервативности: она проходит выше почти во всем диапазоне вариантов.

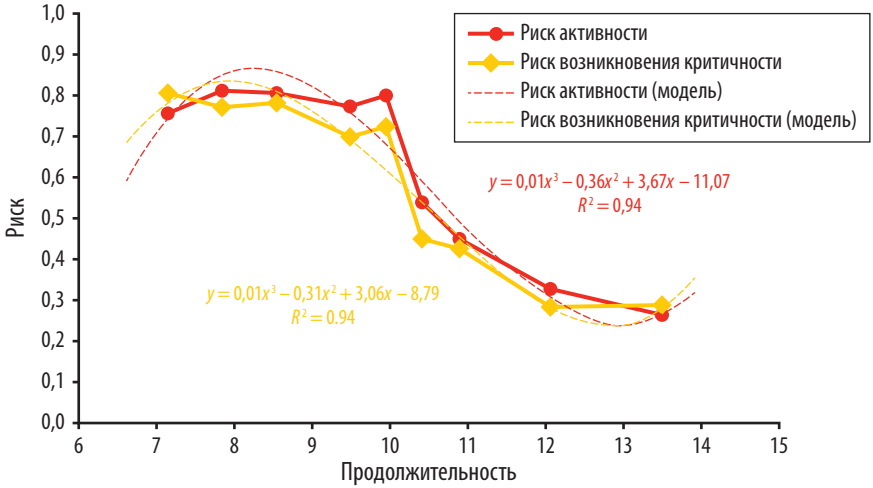


Рис. 11.32. Линии тренда «время-риск» для проекта

В результате подгонки полиномиальной корреляционной модели мы получаем новую формулу риска в проекте:

$$R = 0{,}01t^3 - 0{,}36t^2 + 3{,}67t - 11{,}07,$$

где t измеряется в месяцах.

Зная формулу риска, можно построить модель риска на одной диаграмме с моделью прямых затрат, как на рис. 11.33.

Рис. 11.33. Модели риска и прямых затрат

ПРИМЕЧАНИЕ Малый коэффициент при первом слагаемом многочлена в сочетании с высокой степенью (третьей) означает, что для этой формулы риска требуется большая точность. Хотя в тексте это не показано, вычисления этой главы используют восемь знаков в дробной части.

Минимум прямых затрат и риска

Как упоминалось ранее, минимум модели прямых затрат составляет 10,8 месяца. Подставляя это значение в формулу риска, вы получаете значение риска 0,52; таким образом, риск в точке минимума прямых затрат равен 0,52. На рис. 11.33 эти значения изображены пунктирными синими линиями.

В главе 10 говорилось о том, что в идеале минимальные прямые затраты должны достигаться при риске 0,5 и что эта точка является рекомендуемой целью разуплотнения. В примере смещение от этой точки составляет 4%. Хотя у этого проекта нет решения с планом проектирования, продолжительность которого составляет ровно 10,8 месяца, известная точка разуплотнения D3 располагается достаточно близко с продолжительностью 10,9 месяца (пунктирные красные линии на рис. 11.33). В практическом смысле эти точки идентичны.

Оптимальный вариант планирования проекта

Значение модели риска в точке D3 равно 0,50; это означает, что D3 также является идеальной целью разуплотнения, одновременно являясь точкой минимума прямых затрат. Это делает D3 оптимальной точкой в отношении прямых затрат, продолжительности и риска. Общие затраты в D3 составляют всего 63,8 человеко-месяца — практически так же, как при минимуме общих затрат. Это также делает D3 оптимальной точкой в отношении общих затрат, продолжительности и риска.

Оптимальность точки означает, что вариант планирования проекта имеет наивысшую вероятность выполнения обязательств плана (само определение успеха). Вы всегда должны стремиться к тому, чтобы ваше проектирование было основано на точке минимума прямых затрат. На рис. 11.34 наглядно представлены временные резервы сети проекта в точке D3. Как видно из диаграммы, сеть выглядит вполне благополучно.

Минимальный риск

По формуле риска также можно вычислить точку минимального риска. Эта точка имеет продолжительность 12,98 месяца, а значение риска в ней равно 0,245. В главе 10 объяснялось, что минимальное значение риска для модели риска возникновения критичности равно 0,25 (с весами [1, 2, 3, 4]). Хотя зна-

чение 0,248 очень близко к 0,25, оно было получено по формуле риска активности, которая, в отличие от модели риска возникновения критичности, не зависит от выбора весов.

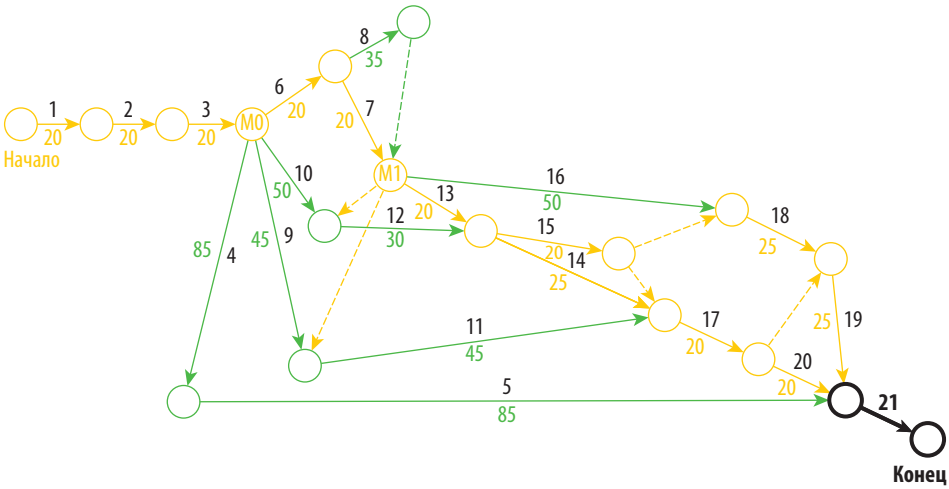


Рис. 11.34. Анализ временных резервов для оптимального решения

Включение и исключение рисков

Из дискретной кривой риска на рис. 11.29 следует, что хотя уплотнение сокращает проект, оно не обязательно приводит к значительному возрастанию риска. Уплотнение этого проекта даже привело к некоторому сокращению риска на кривой риска активности. Основное возрастание риска было обусловлено смещением влево от D3 (или минимальных прямых затрат), а все уплотненные решения обладают высоким риском.

Рисунок 11.35 показывает, как разные решения по планированию проекта соответствуют кривой риска проекта. Мы видим, что второе уплотненное решение дает почти максимальный риск, а для более уплотненных решений характерно ожидаемое снижение уровня риска (эффект да Винчи, представленный в главе 10).

Очевидно, проектировать что-либо в точке максимального риска или после нее не рекомендуется. Следует избегать даже приближения к точке максимального риска для проекта — но где находится эта точка отсечения? В нашем примере максимальное значение риска на кривой риска равно 0,85, так что решения, приближающиеся к этому значению, не могут считаться хорошими вариантами.

В главе 10 значение 0,75 предложено в качестве максимального уровня риска для любого решения. Если риск превышает 0,75, проект обычно становится слишком ненадежным, а график с большой вероятностью будет нарушен.

По формуле риска можно определить, что точка с риском 0,75 имеет продолжительность 9,49 месяца. Хотя ни одно решение не совпадает с этой точкой, первая точка уплотнения имеет продолжительность 9,5 месяца и риск 0,75.

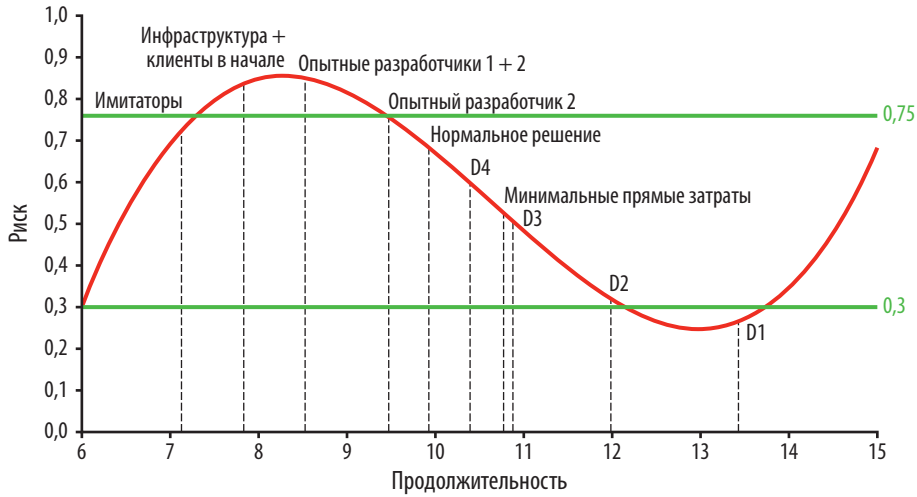


Рис. 11.35. Все решения по планированию проекта и риски

Это позволяет предположить, что первое уплотнение является верхним практическим пределом в данном примере. Как упоминалось ранее, значение 0,3 должно быть минимальным уровнем риска, что исключает точку разуплотнения D1 с риском 0,27. Точка разуплотнения D2 с риском 0,32 возможна, но находится близко к границе.

Анализ SDP

Вся работа по планированию проекта завершается анализом SDP, на котором вы представляете варианты планирования проекта лицам, ответственным за принятие решений. Вы должны не только способствовать принятию обоснованных решений, но и сделать правильный выбор очевидным. Лучшим вариантом проекта до настоящего момента был вариант D3 — одномесячное разуплотнение, обеспечивающее как минимальные затраты, так и риск 0,50.

Представляя результаты ответственным за принятие решений, укажите первую точку уплотнения, нормальное решение и оптимальное месячное разуплотнение от нормального решения. В табл. 11.10 приведена сводка этих потенциально возможных вариантов планирования проекта.

Таблица 11.10. Возможные варианты планирования проекта

Вариант	Продолжительность (месяцы)	Общие затраты (человеко-месяцы)	Риск
Один опытный разработчик	9,5	61,1	0,75
Нормальное решение	9,9	61,1	0,68
Разуплотнение на 1 месяц	10,9	63,8	0,50

Представление вариантов

Я не рекомендую представлять низкоуровневую информацию из табл. 11.10. Вряд ли кто-нибудь из принимающих решения сталкивался с таким уровнем точности, поэтому вашим данным будет не хватать достоверности. Также стоит добавить субкритическое решение. Поскольку субкритическое решение в конечном итоге обходится дороже (а также обладает более высоким риском и занимает больше времени), такие возможности стоит отсекать как можно ранее.

В табл. 11.11 перечислены варианты, которые следовало бы представить на анализе SDP. Обратите внимание на округленные значения сроков и затрат. Округление выполнялось с небольшим намерением создать более очевидные расхождения между вариантами. Хотя это ничего не изменит в процессе принятия решений, числа выглядят более достоверно.

Таблица 11.11. Варианты планирования проекта для анализа

Вариант	Продолжительность (месяцы)	Общие затраты (человеко-месяцы)	Риск
Один опытный разработчик	9	61	0,75
Нормальное решение	10	62	0,68
Разуплотнение на 1 месяц	11	64	0,50
Субкритическое комплектование	13	77	0,78

Значения риска не округляются, потому что риск — лучший показатель для оценки вариантов. Можно быть практически уверенным в том, что ответственные за принятие решений никогда не видели выражения риска в числовой форме как средства для принятия обоснованных решений. Вы должны объяснить, что значения риска нелинейны; другими словами, с числами из табл. 11.11 риск 0,68 *намного* выше риска 0,5, а не просто выше на 36%. Для демонстрации нелинейности можно воспользоваться аналогией между риском и более знакомыми нелинейными характеристиками — например, шкалой Рихтера для силы землетрясений. Если бы показатели риска были уровнями землетрясений по шкале Рихтера, то землетрясение силой 6,8 балла было бы в 500 раз мощнее землетрясения силой 5,0 балла, а землетрясение силой 7,5 балла было бы мощнее в 5623 раза. Простые аналогии такого рода направляют решение к желательной точке с риском 0,50.

12 Расширенные методы планирования

Планирование проекта — весьма нетривиальная дисциплина, и в предыдущих главах рассматривались только базовые концепции. Это было сделано намеренно, чтобы не создавать избыточных сложностей на начальном этапе. Однако область планирования проекта этим далеко не ограничивается, и в этой главе будут представлены дополнительные методы, которые принесут пользу практически в любом проекте, а не только в самых крупных или сложных. У всех этих методов есть кое-что общее: они позволяют лучше управлять риском и сложностью. Вы также увидите, как успешно справляться даже к самыми сложными и проблематичными проектами.

Божественные активности

Как следует из названия, божественные активности слишком велики для вашего проекта. Возможно, термин «слишком велики» нужно рассматривать как относительный — божественная активность слишком велика в отношении других активностей в проекте. Простой критерий для выявления божественных активностей — наличие активности, продолжительность которой отличается от средней продолжительности всех активностей проекта как минимум на одно стандартное отклонение. Впрочем, божественные активности могут быть слишком большими и в абсолютном отношении. Продолжительность в диапазоне 40–60 дней (и более) слишком велика для типичного программного проекта.

Возможно, ваши интуиция и опыт уже подсказывают вам, что от таких активностей стоит держаться подальше. Как правило божественные активности становятся обычными заполнителями для некой великой неопределенности, скрывающейся под ними. Оценки продолжительности и трудозатрат для божественной активности почти всегда имеют низкую точность. Соответственно, фактическая продолжительность может оказаться длиннее — теоретически достаточно для того, чтобы загубить проект. С такими опасностями следует

разбираться как можно раньше, чтобы у вас сохранялись шансы выполнения своих обязательств.

Божественные активности также нередко искажают методы планирования проекта, описанные в книге. Они почти всегда являются частью критического пути, в результате чего многие методы управления критическим путем становятся неэффективными, потому что продолжительность критического пути и его положение в сети смещаются к божественным активностям. Ситуация усугубляется тем, что модели риска для проектов с божественными активностями выдают обманчиво низкие значения рисков. Большая часть усилий по такому проекту будет потрачена на критические божественные активности, в результате чего проект во всех практических отношениях станет проектом с высоким риском. Однако вычисление риска даст заниженные результаты, потому что другие активности, окружающие критические божественные активности, будут иметь высокий временной резерв. Если убрать эти окружающие активности, то риск моментально вырастет до 1,0 — правильный признак высокого риска, возникающего при использовании божественных активностей.

Решение проблемы божественных активностей

Лучший план действий с божественными активностями — разбиение их на меньшие независимые активности. Разбиение божественных активностей значительно улучшит качество оценки, сократит неопределенность и предоставит правильное значение риска. Но что, если объем работы будет действительно огромным? Такие активности следует рассматривать как мини-проекты и заняться их уплотнением. Начните с выявления внутренних фаз божественных активностей и поиска возможностей параллельной работы в этих фазах внутри каждой божественной активности. Если это невозможно, попробуйте сделать божественные активности менее критическими, убрав их с пути других активностей в проекте. Например, разработка имитаторов для божественных активностей снижает зависимости других активностей от самих божественных активностей. Это позволит организовать работу параллельно с божественными активностями, что сделает их менее критичными (или вообще некритичными). Имитаторы также сокращают неопределенность божественных активностей, так как устанавливаемые ими ограничения раскрывают скрытые предположения; это упрощает подробное проектирование божественных активностей.

Также следует рассмотреть возможности выделения божественных активностей в самостоятельные второстепенные проекты. Выделение во второстепенный проект особенно важно в том случае, если внутренние фазы божественной активности последовательны по своей природе. Это значительно упрощает управление проектом и отслеживание прогресса. Обязательно спланируйте точки интеграции в сети, чтобы сократить риск интеграции на завершающей

стадии. Выделение божественных активностей подобным образом обычно повышает риск для остальной части проекта (после извлечения божественных активностей на долю других активностей остается намного меньший временной резерв). Обычно это хорошо, потому что в противном случае проект имел бы обманчиво низкие показатели риска. Ситуация встречается настолько часто, что низкие показатели риска часто указывают на то, что вам стоит заняться поиском божественных активностей.

Точка пересечения риска

В примере из главы 11 для включения и исключения вариантов планирования проекта использовались простые правила: риск должен быть ниже 0,75 и выше 0,3. При принятии решений о вариантах планирования можно действовать с большей точностью, не ограничивающейся простейшими правилами. На рис. 11.33 в точке минимума прямых затрат и непосредственно слева от нее кривая прямых затрат практически горизонтальна, но кривая риска имеет значительный угол наклона. Такое поведение ожидаемо, потому что кривая риска обычно достигает своего максимального значения до того, как прямые затраты достигнут своего максимума в решениях с наибольшим уплотнением. Единственная возможность достичь максимального риска до максимума прямых затрат — если изначально слева от минимума прямых затрат кривая риска растет намного быстрее кривой прямых затрат. В точке максимального риска (и немного справа от нее) кривая риска горизонтальная или почти горизонтальная, тогда как кривая прямых затрат имеет достаточно крутой угол наклона.

Отсюда следует, что слева от минимума прямых затрат должна существовать точка, в которой кривая риска прекращает расти быстрее кривой прямых затрат. Я называю эту точку *точкой пересечения риска*. В этой точке риск достигает максимума. Отсюда следует, что вам, вероятно, следует избегать уплотненных решений со значениями риска выше точки пересечения. В большинстве проектов точка пересечения риска совпадает со значением 0,75 на кривой риска.

Точка пересечения риска консервативна, потому что она не находится на максимуме риска, и при этом она базируется на поведении риска и прямых затрат, а не на абсолютном значении риска. Отсюда следует, что с учетом истории многих программных проектов некоторая доля осторожности никогда не повредит.

Вычисление точки пересечения

Для нахождения точки пересечения риска потребуется сравнить скорость роста кривой прямых затрат и кривой риска. Это можно сделать разными способами: аналитически — с применением базовых средств математического анализа; графически — с использованием электронной таблицы; или средствами решения числовых уравнений. В файлах, прилагаемых к этой главе, все три

метода применяются почти по шаблонной схеме, так что вы сможете легко найти точку пересечения риска.

Скорость роста функции выражается ее первой производной, поэтому вы должны сравнить первую производную кривой риска с первой производной кривой прямых затрат. Модель риска в примере из главы 11 имеет форму кубического многочлена следующего вида:

$$y = ax^3 + bx^2 + cx + d.$$

Первая производная этой функции представляет собой квадратный многочлен:

$$y' = 3ax^2 + 2bx + c.$$

В нашем примере формула риска выглядит так:

$$R = 0{,}01t^3 - 0{,}36t^2 + 3{,}67t - 11{,}07.$$

Первая производная риска:

$$R' = 0{,}03t^2 - 0{,}72t + 3{,}67.$$

Формула прямых затрат:

$$C = 0{,}99t^2 - 21{,}32t + 136{,}57.$$

Следовательно, первая производная прямых затрат выглядит так:

$$C' = 1{,}98t - 21{,}32.$$

Перед сравнением формул двух производных необходимо решить две проблемы. Во-первых, диапазоны значений между максимальным риском и минимальными прямыми затратами на обеих кривых монотонно убывают (что означает, что скорости роста двух кривых будут отрицательными), поэтому сравнивать нужно абсолютные значения скоростей роста. Во-вторых, исходные скорости роста несравнимы по величине. Значения риска лежат в диапазоне от 0 до 1, а значения затрат в нашем примере равны приблизительно 30. Для корректного сравнения двух производных необходимо сначала масштабировать значения риска до значений затрат в точке максимального риска.

Рекомендуемый коэффициент масштабирования определяется по формуле:

$$F = \frac{R(t_{mr})}{C(t_{mr})},$$

где:

- t_{mr} — время достижения максимального риска;
- $R(t_{mr})$ — значение формулы риска проекта в точке t_{mr};
- $C(t_{mr})$ — значение формулы затрат проекта в точке t_{mr}.

Кривая риска достигает максимума в точке, в которой первая производная кривой риска R' равна 0. Решая уравнение риска для t при $R' = 0$, получаем значение t_{mr}, равное 8,3 месяца. Соответствующее значение риска R равно 0,85, а соответствующее значение прямых затрат — 28 человеко-месяцев. Отношение двух значений F равно 32,93 — это и есть коэффициент масштабирования для нашего примера.

Допустимый уровень риска для проекта достигается при выполнении всех условий из следующего списка:

- Значение времени находится слева от точки минимального риска проекта.
- Значение времени находится справа от точки максимального риска проекта.
- Риск растет быстрее затрат в абсолютном значении с учетом масштабирования.

Эти условия можно объединить в следующем выражении:

$$F \times |R'| > |C'|.$$

Используя уравнения производных риска и прямых затрат, а также коэффициента масштабирования, получаем:

$$32{,}93 \times |0{,}03t^2 - 0{,}72t + 3{,}67| > |1{,}98t - 21{,}32|.$$

Решение уравнения дает допустимый диапазон для t:

$$9{,}03 < t < 12{,}31.$$

В результате получаем не одну, а две точки пересечения — 9,03 и 12,31 месяца.

На рис. 12.1 наглядно представлено поведение масштабированных производных риска и затрат в абсолютных значениях. Из диаграммы хорошо видно, что производная риска в абсолютном значении пересекает производную затрат в абсолютном значении в двух местах (отсюда и термин «точка пересечения»).

Если отложить в сторону математику, существование двух точек пересечения риска связано с семантикой точек с позиций планирования проекта. В точке

9,03 месяца риск равен 0,81; в точке 12,31 месяца риск равен 0,28. Наложение этих значений на кривую риска и кривую прямых затрат на рис. 12.2 раскрывает истинный смысл точек пересечения.

Решения планирования проекта слева от 9,03-месячной точки пересечения риска слишком рискованны; решения справа от 12,31-месячной точки слишком безопасны. Между двумя точками пересечения риска уровень риска «в самый раз».

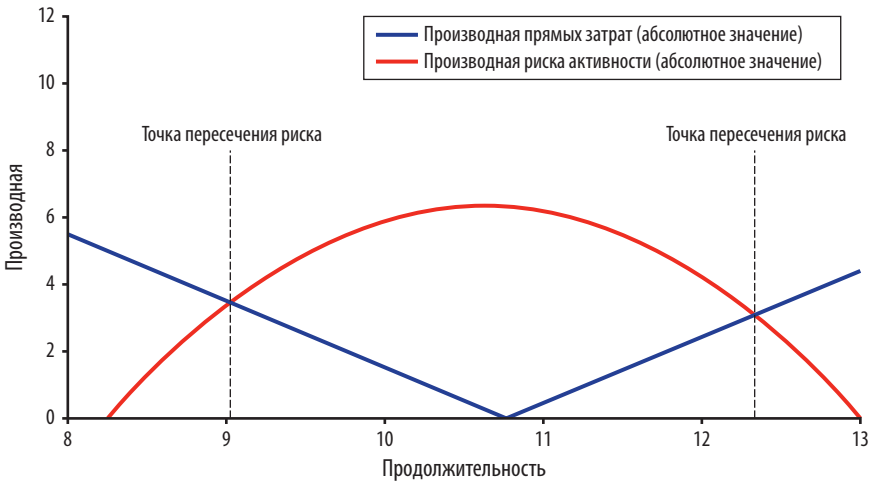


Рис. 12.1. Точки пересечения риска

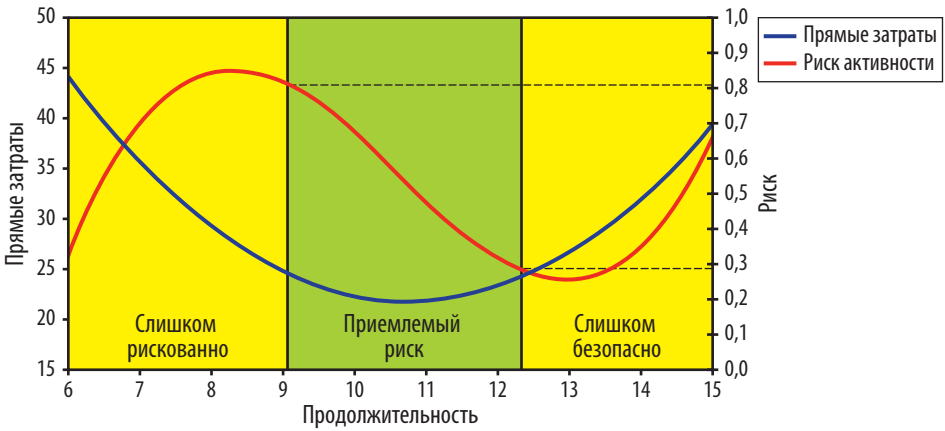


Рис. 12.2. Зоны включения и исключения риска

Приемлемые риски и варианты планирования

Значения риска в точках пересечения 0,81 и 0,28 хорошо согласуются с эмпирическими порогами 0,75 и 0,30. В нашем примере зона приемлемого риска включает первое уплотненное решение, нормальное решение и точки разуплотнения D4, D3 и D2 (рис. 11.35). Все эти точки являются практичными вариантами планирования. «Практичность» в этом контексте означает, что обязательства по проекту будут выполнены с разумной вероятностью. Более уплотненные решения слишком рискованны, а точка D1 слишком безопасна. Вы также можете выбрать между несколькими точками разуплотнения, определив лучшую цель для разуплотнения.

Поиск цели для разуплотнения

Как указано в главе 10, уровень риска 0,5 является точкой наибольшей крутизны на кривой риска. Это делает ее идеальной целью разуплотнения, потому что она обеспечивает наилучший результат — иначе говоря, за минимальную величину разуплотнения вы получаете наибольшее сокращение. Идеальная точка является точкой перегиба риска, а следовательно, минимальной точкой разуплотнения.

Если вы построили график кривой риска, то уже видите, где находится точка перегиба, и можете выбрать точку разуплотнения в точке перегиба или, если действовать более консервативно, справа от нее. Этот метод использовался в главе 11 для того, чтобы рекомендовать D3 на рис. 11.29 в качестве цели разуплотнения. Однако простое определение положения точки на глаз вряд ли можно признать хорошей инженерной практикой. Вместо этого следует применить методы математического анализа для последовательного и объективного поиска цели разуплотнения.

С учетом того, что кривая риска эмулирует стандартную логистическую функцию (по крайней мере между точками минимального и максимального риска), точка кривой с наибольшим углом наклона также отмечает точку перегиба кривой. Слева от этой точки кривая риска является вогнутой, справа — выпуклой. Из математического анализа известно, что в такой точке вторая производная функции равна нулю. Идеальная кривая риска и первые две ее производные показаны в графическом виде на рис. 12.3.

Используя пример из главы 11 для демонстрации этого метода, мы получаем формулу риска в виде кубического многочлена. Первая и вторая производные:

$$y = ax^3 + bx^2 + cx + d;$$

$$y' = 3ax^2 + 2bx + c;$$

$$y'' = 6ax + 2b.$$

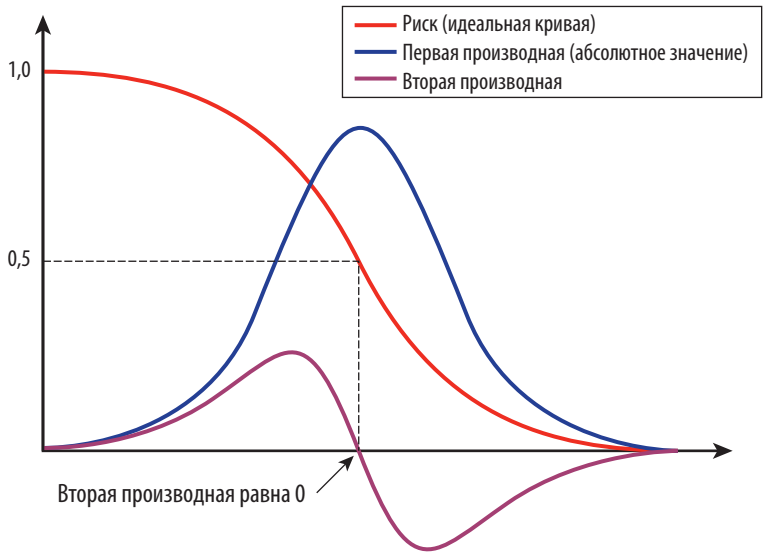


Рис. 12.3. Точка перегиба как цель разуплотнения

Приравнивая вторую производную к нулю, получаем следующую формулу:

$$x = -\frac{b}{3a}.$$

Так как модель риска выглядит так:

$$R = 0{,}01t^3 - 0{,}36t^2 + 3{,}67t - 11{,}07,$$

нулевое значение второй производной достигается в точке 10,62 месяца:

$$t = \frac{-0{,}36}{3 \times 0{,}01} = 10{,}62.$$

В точке 10,62 месяца значение риска равно 0,55, что только на 10% отличается от идеальной цели 0,5. При нанесении на дискретные кривые риска на рис. 12.4 становится видно, что это значение находится в диапазоне между D4 и D3; таким образом обосновывается выбор D3 в качестве цели разуплотнения в главе 11.

В отличие от главы 11, в которой использовалось наглядное представление диаграммы рисков, а точка перегиба определялась субъективно, вторая производная предоставляет объективный и воспроизводимый критерий. Это особенно важно при отсутствии очевидной точки перегиба, а также при смещении кривой риска вверх или вниз, когда рекомендованное значение 0,5 не подходит.

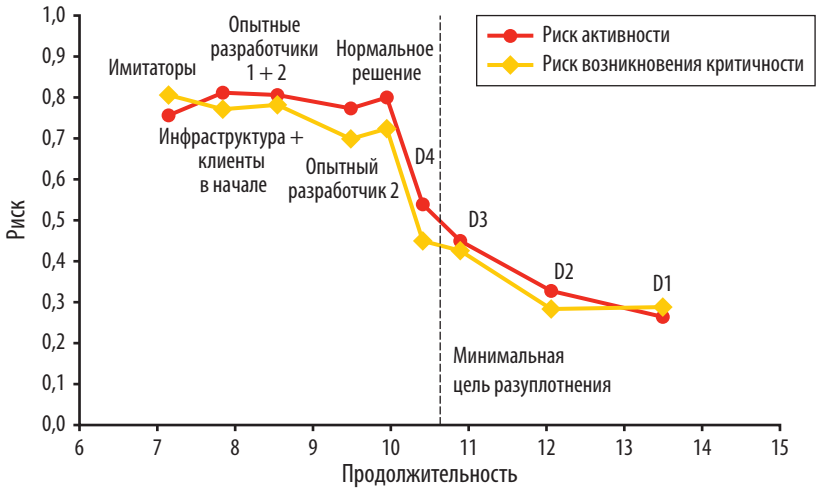


Рис. 12.4. Цель разуплотнения на кривых рисков

Геометрический риск

Все модели риска, представленные в главе 10, используют для вычисления риска разновидность среднего арифметического временных резервов. К сожалению, среднее арифметическое плохо справляется с неравномерными распределениями значений. Возьмем последовательность [1, 2, 3, 1000]. Среднее арифметическое этой последовательности равно 252, что совершенно не типично для имеющихся значений. Такое поведение проявляется не только при вычислении рисков, а любые попытки использования среднего арифметического для крайне неравномерного распределения приводят к неудовлетворительному результату. В таком случае лучше использовать среднее геометрическое вместо среднего арифметического.

Среднее геометрическое для последовательности значений вычисляется умножением всех n значений последовательности и последующим извлечением корня n-й степени из произведения. Для последовательности значений $a_1 \ldots a_n$ среднее геометрическое последовательности вычисляется по следующей формуле:

$$\text{Среднее} = \sqrt[n]{a_1 \times a_2 \times \ldots \times a_n} = \sqrt[n]{\prod_{i-1}^{n} a_i}.$$

Например, хотя среднее арифметическое последовательности [2, 4, 6] равно 4, среднее геометрическое равно 3,63:

$$\text{Среднее} = \sqrt[3]{2 \times 4 \times 6} = 3{,}63.$$

Среднее геометрическое всегда меньше или равно среднему арифметическому той же последовательности значений:

$$\sqrt[n]{\prod_{i=1}^{n} a_i} \leqslant \frac{\sum_{i=1}^{n} a_i}{n}.$$

Два средних равны только в том случае, когда все значения в последовательности одинаковы.

Хотя на первый взгляд может показаться, что среднее геометрическое — какая-то алгебраическая странность, оно по-настоящему проявляет себя в последовательностях с неравномерным распределением значений. При вычислении среднего геометрического выбросы намного меньше влияют на результат. В примере с последовательностью [1, 2, 3, 1000] среднее геометрическое равно 8,8, что гораздо лучше представляет первые три числа последовательности.

Геометрический риск возникновения критичности

Как и в случае с арифметическим риском возникновения критичности, для вычисления геометрического риска возникновения критичности можно воспользоваться цветовым кодированием временных резервов и соответствующим количеством активностей. Вместо того чтобы умножать веса временных резервов на количество активностей, вы возводите их в эту степень. Геометрическая формула риска возникновения критичности выглядит так:

$$\text{Риск} = \frac{\sqrt[N]{(W_C)^{N_C} \times (W_R)^{N_R} \times (W_Y)^{N_Y} \times (W_G)^{N_G}}}{W_C},$$

где:

- W_C — вес критических активностей;
- W_R — вес красных активностей;
- W_Y — вес желтых активностей;
- W_G — вес зеленых активностей;
- N_C — количество критических активностей;
- N_R — количество красных активностей;
- N_Y — количество желтых активностей;
- N_G — количество зеленых активностей;
- N — количество активностей в проекте ($N = N_C + N_R + N_Y + N_G$).

Для сети на рис. 10.4 геометрический риск возникновения критичности выглядит так:

$$\text{Риск} = \frac{\sqrt[16]{4^6 \times 3^4 \times 2^2 \times 1^4}}{4} = 0{,}60.$$

Соответствующий арифметический риск возникновения критичности для той же сети равен 0,69. Как и ожидалось, геометрический риск несколько ниже арифметического.

Диапазон значений риска

Как и в случае с арифметическим риском, геометрический риск возникновения критичности имеет максимальное значение 1,0, когда все активности критичны, и минимальное значение W_G/W_C, когда все активности сети являются зелеными:

$$\text{Риск} = \frac{\sqrt[N]{(W_C)^0 \times (W_R)^0 \times (W_Y)^0 \times (W_G)^N}}{W_C} = \frac{\sqrt[N]{1 \times 1 \times 1 \times (W_G)^N}}{W_C} = \frac{W_G}{W_C}.$$

Геометрический риск Фибоначчи

Для получения геометрической модели риска Фибоначчи можно использовать отношение Фибоначчи между весами. Для следующего определения весов:

$$W_Y = \varphi \times W_G;$$
$$W_R = \varphi^2 \times W_G;$$
$$W_C = \varphi^3 \times W_G$$

геометрическая формула Фибоначчи выглядит так:

$$\text{Риск} = \frac{\sqrt[N]{(\varphi^3 \times W_G)^{N_C} \times (\varphi^2 \times W_G)^{N_R} \times (\varphi \times W_G)^{N_Y} \times (W_G)^{N_G}}}{\varphi^3 \times W_G} =$$
$$= \frac{\sqrt[N]{(\varphi^{3N_C+2N_R+N_Y} \times W_G^{N_C+N_R+N_Y+N_G}}}{\varphi^3 \times W_G} = \frac{\sqrt[N]{(\varphi^{3N_C+2N_R+N_Y} \times W_G^{N}}}{\varphi^3 \times W_G} = \frac{\sqrt[N]{(\varphi^{3N_C+2N_R+N_Y}}}{\varphi^3} =$$
$$= \varphi^{\frac{3N_C+2N_R+N_Y}{N}-3}.$$

Диапазон значений риска

Как и в случае с арифметическим риском Фибоначчи, геометрический риск Фибоначчи имеет максимальное значение 1,0, когда все активности критичны,

и минимальное значение 0,24 (φ^{-3}), когда все активности сети являются зелеными.

Геометрический риск активности

Формула геометрического риска активности использует среднее геометрическое временных резервов в проекте. Критические активности имеют нулевой временной резерв, и это создает проблему, потому что среднее геометрическое всегда будет равно нулю. Стандартное обходное решение — прибавить 1 ко всем значениям последовательности и вычесть 1 из полученного среднего геометрического.

Таким образом, формула геометрического риска активности имеет вид:

$$\text{Риск} = 1 - \frac{\sqrt[N]{\prod_{i=1}^{N}(F_i + 1)} - 1}{M},$$

где:

- F_i — временной резерв для активности i;
- N — количество активностей в проекте;
- M — максимальный временной резерв по всем активностям в проекте, или $\text{Max}(F_1, F_2, ..., F_N)$.

Для сети на рис. 10.4 геометрический риск активности будет равен:

$$\text{Риск} = 1 - \frac{\sqrt[16]{1\times1\times1\times1\times1\times1\times31\times31\times31\times31\times11\times11\times6\times6\times6\times6} - 1}{30} = 0{,}87.$$

Соответствующий арифметический риск активности для той же сети будет равен 0,67.

Диапазон значений риска

Максимальное значение геометрической модели риска активности приближается к 1,0 по мере того, как все больше активностей приближается к критическому состоянию, но становится неопределенным, когда критическими становятся все активности. Геометрический риск активности имеет минимальное значение 0, когда временной резерв всех активностей одинаков. В отличие от арифметического риска активности, с геометрическим риском активности нет необходимости корректировать выбросы с аномально высоким временным резервом, и временные резервы могут не иметь равномерного распределения.

Поведение геометрического риска

Обе модели — как геометрического риска возникновения критичности, так и геометрического риска Фибоначчи — дают результаты, сходные с результатами их арифметических аналогов. При этом формула геометрической активности отличается от своего арифметического аналога и дает заметно большие значения в диапазоне. В результате значения геометрического риска активности обычно не соответствуют рекомендациям по значениям риска, приведенным в книге.

На рис. 12.5 представлены различия в поведении между моделями геометрических рисков, для чего на диаграмму нанесены все кривые риска из примера главы 11.

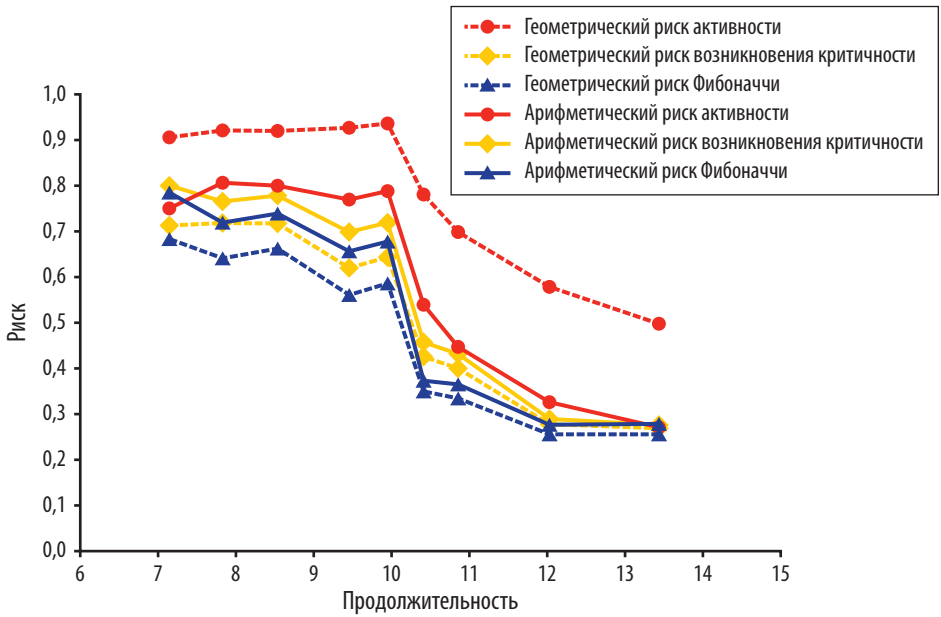


Рис. 12.5. Геометрические и арифметические модели риска

Как видите, геометрический риск возникновения критичности и геометрический риск Фибоначчи имеют ту же общую форму, что и арифметические модели, только проходят чуть ниже, как и ожидалось. На диаграмме хорошо видна та же точка перегиба риска. Геометрический риск активности заметно приподнят, а его поведение сильно отличается от поведения арифметического риска активности. Хорошо заметная точка перегиба на нем отсутствует.

Для чего нужен геометрический риск?

Почти идентичное поведение арифметической и геометрической моделей риска возникновения критичности (а также модели Фибоначчи) демонстрирует,

что не так уж важно, какую именно модель вы будете использовать. Различия могут не оправдать времени и усилий, потраченных на построение очередной кривой риска для проекта. Пожалуй, для простоты при объяснении моделирования рисков другим предпочтение стоит отдавать арифметической модели. Геометрический риск активности бесспорно менее полезен, чем арифметический риск активности, но его полезность в одной конкретной ситуации — та причина, по которой я решил обсудить геометрический риск.

Геометрический риск активности — последняя мера при попытке вычислить риск проекта с божественной активностью. Такой проект обладает очень высоким риском, потому что большая часть усилий тратится на критические божественные активности. Как объяснялось ранее, из-за размера божественных активностей другие активности обладают значительным временным резервом, что, в свою очередь, приводит к занижению арифметического риска и создает ложное чувство безопасности. Напротив, модель геометрического риска активности предоставляет ожидаемое высокое значение риска для проектов с божественными активностями. Для геометрического риска активности можно построить корреляционную модель и провести тот же анализ риска, что и для арифметической модели.

На рис. 12.6 показан геометрический риск активности и его корреляционная модель для проекта, описанного в главе 11.

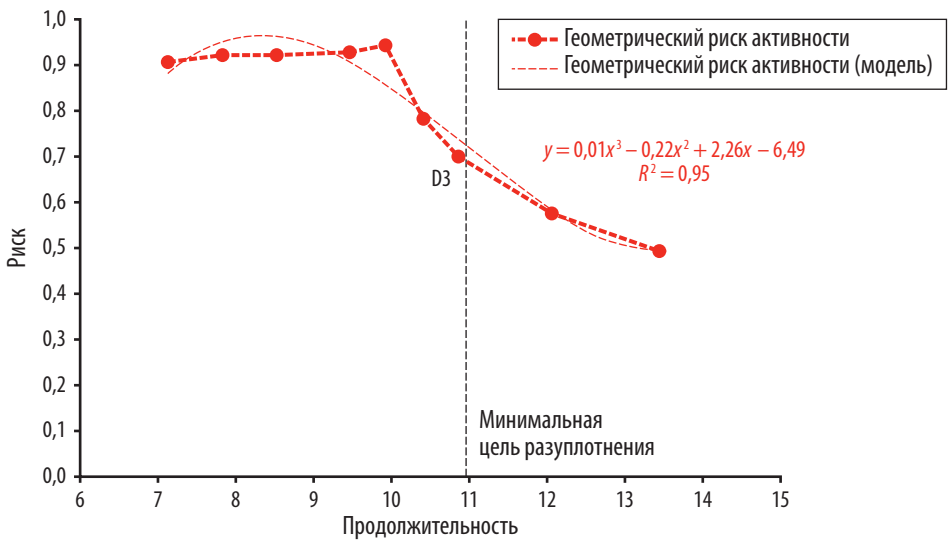


Рис. 12.6. Геометрическая модель риска активности

Точка максимального риска (8,3 месяца) является общей как для арифметических, так и для геометрических моделей. Минимальная цель разуплотнения для

модели геометрической активности (в которой производная равна нулю) находится в точке 10,94 месяца — близко к точке 10,62 месяца для арифметической модели и непосредственно справа от D3. Точками пересечения для геометрического риска являются точки 9,44 и 12,25 месяца — чуть более узкий диапазон по сравнению с 9,03 и 12,31 месяца, полученными при использовании арифметической модели риска активности. Как видите, результаты двух моделей в целом похожи, несмотря на заметные различия в поведении кривых риска.

Конечно, вместо того чтобы искать возможности вычисления риска в проекте с божественными активностями, следует решить проблему с божественными активностями так, как описано выше. Однако геометрический риск позволяет разобраться с ситуацией в том виде, в котором она есть, а не в том, в котором она должна быть.

Сложность исполнения

В предыдущих главах обсуждение планирования проекта было направлено на принятие обоснованных решений до начала работы. Только на основании полученных числовых метрик продолжительности, затрат и рисков можно решить, жизнеспособен ли проект. Однако два варианта планирования проекта могут совпадать по продолжительности, затратам и риску, но при этом сильно различаться по сложности исполнения. Говоря о сложности исполнения, я в данном случае имею в виду то, насколько запутанна и проблематична сеть проекта.

Цикломатическая сложность

Цикломатическая сложность является метрикой сложности сетевых связей. Она может использоваться для оценки сложности всего, что может быть выражено в форме сети, включая программный код и проекты.

Формула цикломатической сложности:

$$\text{Сложность} = E - N + 2 \times P.$$

Для сложности исполнения проекта:

- E — количество зависимостей в проекте.
- N — количество активностей в проекте.
- P — количество компонент связности в проекте.

В хорошо спроектированной сети значение P всегда равно 1, потому что проект должен состоять из одной сети. Множественные сети ($P > 1$) сделают проект более сложным.

Продемонстрируем формулу цикломатической сложности для сети из табл. 12.1: *E* равно 6, *N* равно 5, а *P* равно 1. Цикломатическая сложность равна 3:

$$\text{Сложность} = 6 - 5 + 2 \times 1 = 3.$$

Таблица 12.1. Пример сети с цикломатической сложностью 3

ID	Активность	Зависит от
1	A	
2	B	
3	C	1, 2
4	D	1, 2
5	E	3, 4

Тип проекта и сложность

Хотя не существует механизма прямой оценки сложности исполнения проекта, вы можете воспользоваться формулой цикломатической сложности в качестве заместителя. Чем больше внутренних зависимостей содержит проект, тем более рискованным и сложным становится его исполнение. Любая из этих зависимостей может столкнуться с задержкой, что приведет к каскадным задержкам во многих других местах проекта. Максимальная цикломатическая сложность проекта из *N* активностей имеет порядок N^2 — в таких проектах каждая активность зависит от всех остальных активностей.

В общем случае чем более параллельную структуру имеет проект, тем выше будет сложность его исполнения. Как минимум будет непросто своевременно иметь расширенную комплектацию для всех параллельных активностей. Параллельная работа (а также дополнительная работа, необходимая для обеспечения параллельной работы) увеличивает как рабочую нагрузку, так и размер команды. Большая команда будет менее эффективной, а управление ею создает больше сложностей. Параллельная работа также приводит к повышению цикломатической сложности, потому что при параллельной работе *E* растет быстрее, чем *N*. В крайнем случае проект с *N* активностями, которые стартуют одновременно и завершаются одновременно, в котором каждая активность не зависит от всех остальных активностей, а все активности выполняются параллельно, имеет цикломатическую сложность $N + 2$. Исполнение такого проекта сопряжено с огромным риском.

По тому же принципу чем более последовательную природу имеет проект, тем проще он будет в исполнении. В крайнем случае простейший проект с *N* актив-

ностями представляет собой последовательную цепочку активностей. Такой проект обладает минимально возможной цикломатической сложностью, которая равна 1. Субкритические проекты с минимумом ресурсов часто напоминают такие длинные цепочки активностей. Хотя проектировочный риск такого субкритического проекта высок (приближается к 1,0), риск исполнения очень низок.

Мой опыт показывает, что в хорошо спланированных проектах цикломатическая сложность равна 10 или 12. Хотя этот уровень может показаться низким, следует понимать, что вероятность соблюдения обязательств связана со сложностью исполнения непропорционально. Например, проект с цикломатической сложностью 15 всего на 25% сложнее проекта с цикломатической сложностью 12, но проект с более низкой сложностью может иметь вдвое большую вероятность успешного завершения. Таким образом, высокая сложность исполнения положительно коррелирует с вероятностью неудачи. Чем сложнее проект, тем больше вероятность того, что вы не сможете выполнить свои обязательства. Кроме того, успешная реализация одного сложного проекта не гарантирует, что вам удастся повторить этот успех на другом сложном проекте.

Конечно, теоретически вы можете раз за разом успешно справляться с проектами, имеющими высокую цикломатическую сложность, но для развития таких навыков в организации потребуется время. Для этого необходима основательная архитектура; хороший план проекта с правилами относительно риска; команда, участники которой привыкли работать вместе на максимуме производительности; и высококлассный менеджер проекта, который обращает внимание на все мелочи и активно разрешает конфликты. Если чего-то из этого не хватает, вам следует активно поработать над сокращением сложности исполнения с применением методов проектирования по уровням и сети сетей, описанных далее в этой главе.

Уплотнение и сложность

Сложность обычно возрастает с уплотнением, и чаще всего возрастает нелинейно. В идеале сложность ваших решений по планированию проекта как функция их продолжительности имеет вид пунктирной кривой на рис. 12.7.

Проблема с таким классическим нелинейным поведением заключается в том, что оно не учитывает уплотнение проекта при использовании более опытных ресурсов без изменения сети проекта. Пунктирная линия также предполагает, что сложность может быть дополнительно понижена за счет постоянно возрастающего выделения времени, но, как упоминалось ранее, сложность имеет жестко зафиксированный минимум 1. Улучшенная модель сложности проекта представляет собой некоторую разновидность логистической функции (сплошная линия на рис. 12.7).

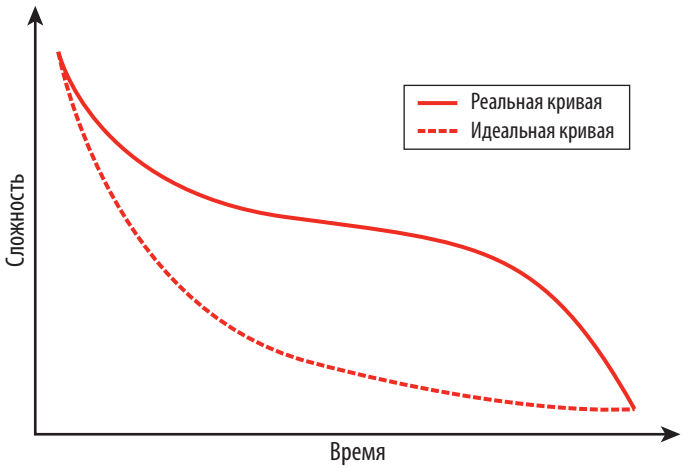


Рис. 12.7. Кривая «время-сложность» проекта

Относительно плоская область логистической функции представляет ситуацию работы с лучшими ресурсами. Резкое возрастание в левой части кривой соответствует параллельной работе и уплотнению проекта. Резкое падение в правой части кривой представляет субкритические решения проекта (которые тоже занимают намного больше времени). На рис. 12.8 это поведение представлено на примере кривой сложности проекта из главы 11.

Рис. 12.8. Кривая «время-сложность» для проекта из примера

Вспомните, о чем говорилось в главе 11: даже самое уплотненное решение не было ощутимо более затратным, чем нормальное решение. Анализ сложности показывает, что истинной ценой максимального уплотнения в данном случае было 25-процентное возрастание цикломатической сложности — признак того, что исполнение проекта становится гораздо более проблемным и рискованным.

Очень большие проекты

Методология планирования проектов, описанная в книге, хорошо работает независимо от масштаба. Тем не менее с увеличением размера проекта задача усложняется. Способность человеческого мозга представлять полную картину со всеми подробностями, ограничениями и взаимными зависимостями внутри проекта не бесконечна. При каком-то размере проекта вы уже не сможете планировать проект. Большинство людей может планировать проект, содержащий около 100 активностей. С опытом это количество может расти. Хорошо спроектированная система и план проекта позволят вам справиться даже с несколькими сотнями активностей.

Мегапроекты со многими сотнями и даже тысячами активностей имеют особый уровень сложности. Обычно в них задействованы разные производственные площадки, десятки и сотни людей, огромные бюджеты и жесткие сроки. Обычно последние три фактора встречаются одновременно, потому что компания сначала принимает обязательства с жесткими сроками, а затем направляет на проект время и деньги, надеясь удержаться в рамках графика.

Чем крупнее проект, тем сложнее становится планирование и тем важнее правильно спланировать проект. Во-первых, чем крупнее проект, тем серьезнее последствия в случае его неудачи. Во-вторых (что важнее), вам приходится с самого начала организовывать параллельную работу, потому что никто не станет ждать его завершения 500 лет — и даже 5 лет, если на то пошло. Что еще хуже, мегапроект находится под сильным давлением с самого первого дня, потому что ставкой в таких проектах становится само будущее компании и многие карьеры. Проекты будут находиться под пристальным вниманием, а руководители будут кружить рядом с вами, словно рой разгневанных шершней.

Мегапроекты почти без исключений оказываются мегапровалами. Между размером и неблагополучным исходом существует прямая связь[1]. Чем крупнее проекты, тем сильнее будут отклонения от обязательств, с более долгими за-

[1] Nassim Nicholas Taleb, Antifragile (Random House, 2012). (На русском: *Талеб Нассим Николас*. Антихрупкость. М.: КоЛибри, 2020. — *Примеч. ред.*)

держками и все более высокими затратами по сравнению с исходным графиком и бюджетом. Мегапроекты — это современные циклопические зиккураты, обреченные на провал.

Сложные системы и непрочность

Тот факт, что большие проекты обречены на провал, — не случайность, а прямой результат их сложности. В этом контексте важно отличать сложное от нетривиального. Многие программные системы нетривиальны, но не сложны. Нетривиальная система может обладать детерминированным поведением, и вы можете точно разобраться в ее внутренних механизмах. Такая система обладает известной воспроизводимой реакцией на конкретные входные данные, а ее поведение в прошлом характеризует поведение в будущем. В отличие от нетривиальных систем, климат, экономика и ваше тело являются сложными системами. Для сложных систем характерно отсутствие понимания внутренних механизмов и невозможность прогнозирования их поведения. Сложное поведение не обязательно обусловлено наличием многочисленных сложных внутренних частей. Например, три тела, движущиеся поблизости друг от друга, образуют сложную недетерминированную систему. Даже простой маятник с подвижной точкой подвеса образует сложную систему. Хотя оба этих примера не являются нетривиальными, они остаются сложными системами.

В прошлом сложные программные системы были ограничены кругом критически важных систем, у которых предметная область была изначально сложной. Из-за повышения связности систем за последние два десятилетия, их разнообразия и масштаба облачных вычислений корпоративные и даже обычные программные системы в наши дни проявляют некоторые аспекты поведения сложных систем.

Фундаментальное свойство сложных систем заключается в том, что они нелинейно реагируют на незначительные изменения условий. Речь идет о так называемом эффекте последней снежинки, когда всего одна дополнительная снежинка вызывает сход лавины на заснеженном горном склоне.

Одна снежинка создает такой риск, потому что сложность растет в нелинейной зависимости от размера. В больших системах рост сложности приводит к соизмеримому росту риска неудачи. Сама функция риска может быть сильно нелинейной функцией сложности, сходной с экспоненциальной функцией. Даже если основание функции близко к 1, а размер системы растет медленно (по одной строке кода или по одной снежинке на заснеженный склон), со временем рост сложности и его кумулятивный эффект для риска приведут к провалу, как при вышедшей из-под контроля ядерной реакции.

Факторы сложности

Теория сложности[1] старается объяснить, почему сложные системы ведут себя именно так, а не иначе. Согласно теории сложности, все сложные системы обладают четырьмя ключевыми элементами: связностью, разнообразием, взаимодействиями и циклами обратной связи. Любое нелинейное поведение является результатом воздействия этих факторов сложности.

Даже если система очень велика, а ее части не связаны друг с другом, сложность в ее поведении не проявится. В связной системе из n частей сложность связности растет пропорционально n^2 (так называемый закон Меткалфа[2]). Можно даже говорить о сложности связности порядка n^n из-за каскадных эффектов: любое одиночное изменение порождает n изменений, каждое из которых порождает n дополнительных изменений, и т. д.

Система может иметь связные компоненты, но при этом не быть особенно сложной для управления, если эти части дублируются или являются простыми модификациями друг друга. С другой стороны, чем разнообразнее система (например, если в ней задействованы разные команды с разными инструментариями, стандартами программирования или проектирования), тем более сложной будет система и тем выше будет вероятность ошибок. Для примера рассмотрим авиакомпанию, которая использует 20 разных типов самолетов, каждый из которых предназначен для своего рынка и имеет уникальные запчасти, смазочные материалы, летчиков и графики технического обслуживания. Такая сложная система обречена на провал просто из-за разнообразия. Сравните с авиакомпанией, которая использует один тип самолетов, не предназначенный ни для какого конкретного рынка; такая компания может обслуживать любые рынки с любыми пассажирами и дальностью перевозок. Вторая авиакомпания не просто создает меньше проблем с управлением: она более устойчива и может быстрее реагировать на изменения рыночной ситуации. Такие идеи должны сочетаться с преимуществами компоновочного проектирования, описанного в главе 4.

Связной разнородной системой даже можно управлять — при условии, что вы не допускаете интенсивных взаимодействий между компонентами. Такие взаимодействия могут иметь дестабилизирующие непредвиденные последствия по всей системе, часто связанные с такими аспектами, как график, затраты, качество, исполнение, эффективность, надежность, оборот средств, удовлетворенность заказчика, удержание кадров и моральное состояние коллектива. Если не ослаблять эти изменения, они породят новые взаимодействия в форме циклов обратной связи. Такие циклы обратной связи усиливают проблемы до

---

[1] https://ru.wikipedia.org/wiki/Сложная_система

[2] https://wikipedia.tel/Закон_Меткалфа

точки, в которой входные данные или условия состояния, которые не создавали проблем в прошлом, смогут вывести систему из строя.

> **ПРИМЕЧАНИЕ** Эти факторы сложности также объясняют, почему системы с функциональной декомпозицией сложны и неустойчивы. Функциональная декомпозиция обладает таким же разнообразием, как требуемая функциональность по всем заказчикам и моментам времени. Из-за возникающего огромного разнообразия в архитектуре сложность выходит из-под контроля.

Размер, сложность и качество

Другая причина, по которой большие проекты часто завершаются неудачей, связана с качеством. Если сложная система зависит от завершения серии задач (например, серии взаимодействий между сервисами или активностями в проекте), а сбой любой задачи приводит к сбою всей серии, любые проблемы с качеством приводят к серьезным побочным эффектам, даже если компоненты очень просты. Пример такого рода случился в 1986 году, когда уплотнительное кольцо стоимостью 30 центов привело к катастрофе космического челнока стоимостью 3 миллиарда долларов.

Если качество целого зависит от качества всех компонентов, общее качество продукта является произведением качества отдельных элементов.[1] Результатом является сильно нелинейное поведение распада. Предположим, система выполняет сложную задачу, которая состоит из 10 меньших задач, каждая из которых имеет почти идеальное качество 99%. В этом случае совокупное качество составит всего 90% ($0{,}99^{10} = 0{,}904$).

Даже это предположение о 99-процентном качестве или надежности нереалистично, потому что большинство программных модулей никогда не тестируется до 99% всех возможных входных данных, всех возможных взаимодействий всех связанных компонентов, всех возможных циклов обратной связи при изменениях состояния, всех вариантов развертывания и сред заказчика и т. д. Вероятно, реалистичные показатели качества будут ниже. Если каждый модуль был протестирован и получил оценку на уровне 90%, качество системы падает до 35%. Всего 10-процентное снижение качества на компонент снижает общее качество на 65%.

Чем больше компонентов в системе, тем сильнее будет выражен эффект и тем более уязвимой будет система к любым проблемам с качеством. Это объясняет, почему большие проекты часто страдают от низкого качества вплоть до момента, когда они становятся непригодными для практического применения.

---

[1] Michael Kremer, «The O-Ring Theory of Economic Development», *Quarterly Journal of Economics* 108, no. 3 (1993): 551–575.

Сеть сетей

Ключ к успеху больших проектов — подавление факторов сложности за счет сокращения размера проекта. Проект должен рассматриваться как сеть сетей. Вместо одного очень большого проекта вы создаете несколько меньших, менее сложных проектов с гораздо более высокой вероятностью успеха. Затраты при таком подходе обычно возрастают — по крайней мере незначительно, зато вероятность провала заметно снижается.

Одного наличия сети сетей недостаточно; проект должен быть реализуем, то есть его можно каким-то образом построить таким способом. Если проект реализуем, то существует высокая вероятность того, что сети не имеют сильных связей и возможна сегментация на отдельные подсети. В противном случае проект обречен на провал.

После того как у вас появится сеть сетей, вы проектируете и исполняете каждый подпроект точно так же, как любой другой проект.

> **ПРИМЕЧАНИЕ** Интерпретация большой системы как совокупности сегментов, или подсистем (см. главу 3), также является проявлением концепции сети сетей. Каждый сегмент существует сам по себе и имеет намного меньшую сложность, чем система в целом. Совокупность подсистем обладает еще одним очевидным преимуществом: каждая подсистема имеет меньше компонентов, чем система в целом, а следовательно, менее чувствительна к кумулятивному снижению качества, о котором говорилось выше.

Проектирование сети сетей

Так как вы не знаете заранее, возможна ли сегментация и как выглядит сеть сетей, вам придется выполнить предварительный мини-проект, целью которого является определение сети сетей. Единственно правильного способа проектирования сети сетей не существует; обычно есть несколько разных вариантов формы и структуры. Эти возможности вряд ли будут эквивалентными — работать с некоторыми из них будет проще, чем с другими. Вы должны сравнить варианты и выделить различия между ними.

Как обычно при проектировании, проектирование сети сетей должно быть итеративным. Начните с проектирования мегапроекта, а затем разбейте его на более удобные проекты вдоль критического пути и рядом с ним. Ищите точки сопряжения, в которых сети взаимодействуют друг с другом. Эти точки станут отличным местом для начала сегментации. Ищите сопряжения не только для зависимостей, но и для времени: если группа активностей завершается до того, как начнется другая группа, то существует сопряжение во времени даже в том случае, если активности плотно взаимодействуют друг с другом. Более сложный метод основан на поиске сегментации, минимизирующей общую цикло-

матическую сложность сети сетей. В этом случае значение $P > 1$ допустимо для общей сложности, при этом у каждой подсети $P = 1$.

На рис. 12.9 показан пример мегапроекта, а на рис. 12.10 — три полученные независимые подсети.

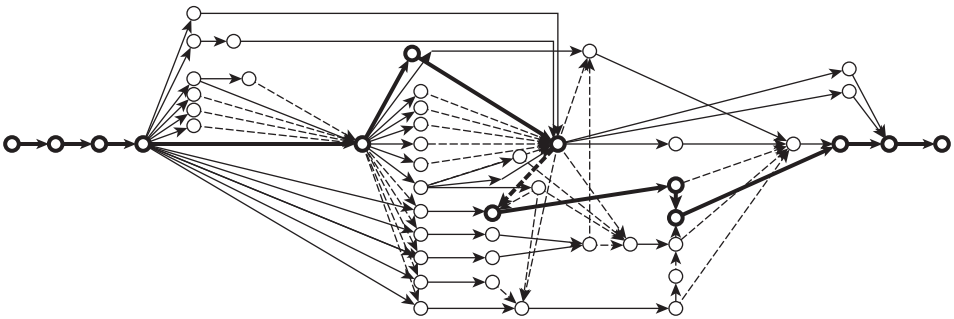


Рис. 12.9. Пример мегапроекта

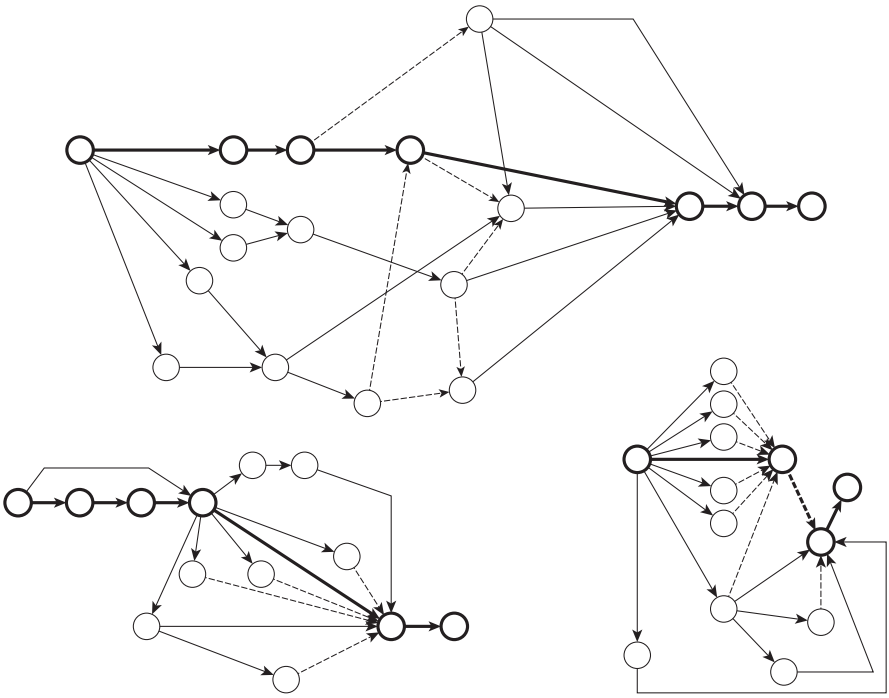


Рис. 12.10. Полученная сеть сетей

Довольно часто исходный мегапроект оказывается слишком запутанным для такой работы. В таком случае выделите время на упрощение или улучшение структуры мегапроекта — это поможет вам определить сеть сетей. Поищите возможности сократить сложность за счет введения предпосылок планирования и установления ограничений для мегапроекта. Примите меры к тому, чтобы некоторые фазы завершались до начала других. Устраните решения, маскирующиеся под требования.

Диаграмма на рис. 12.9 прошла несколько итераций по снижению сложности для достижения показанного состояния. Исходная диаграмма была запутанной и непригодной для работы.

Устранение связей между сетями

Скорее всего, сеть сетей будет включать некоторые зависимости, которые препятствуют сегментации или параллельной работе в разных сетях, по крайней мере на начальной стадии. Для решения этих проблем можно воспользоваться следующими средствами устранения связей между сетями:

- Архитектура и интерфейсы.
- Имитаторы.
- Стандарты разработки.
- Построение, тестирование и автоматизация развертывания.
- Обеспечение качества (не только контроль качества).

Творческие решения

Единственно верной формулы для построения сети сетей не существует. Лучшее, что можно посоветовать, — действовать творчески. Вам часто придется прибегать к творческим решениям нетехнических проблем, мешающих сегментации. Возможно, политические факторы и противодействие приводят к концентрации частей мегапроекта вместо их распределения. В таких случаях необходимо выявить расстановку сил и разрядить кризисную ситуацию, чтобы сегментация стала возможной. Возможно, общеорганизационные проблемы, включая внутреннюю конкуренцию, мешают нормальному взаимодействию и сотрудничеству в сетях, проявляясь в форме жесткого последовательного потока операций в проекте. А может быть, разработчики находятся в разных филиалах, и руководство настаивает на том, чтобы загрузить работой каждый филиал — по функциональному принципу. Такая декомпозиция не имеет никакого отношения к правильной структуре сети сетей или к настоящей квалификации проектировщика. Возможно, вам придется предложить серьезную реорганизацию, включая возможное перемещение персонала, чтобы организа-

ция соответствовала структуре сети сетей, а не наоборот (подробнее об этом в следующем разделе).

Представьте, что какая-то унаследованная группа (*legacy group*) должна стать частью проекта по личным причинам. Вместо сегментации это создаст для проекта узкое место, потому что вся остальная работа теперь начинает вращаться вокруг унаследованной группы. Одно из возможных решений — преобразование унаследованной группы в межсетевую группу инженеров по тестированию, являющихся экспертами в предметной области.

Наконец, опробуйте несколько вариантов структуры сети сетей от разных людей — возможно, кто-то увидит простое решение, которое не заметят другие. Если учесть, что поставлено на карту, вы должны взглянуть на происходящее под всеми возможными углами. Не жалейте времени на то, чтобы тщательно спроектировать сеть сетей. Избегайте спешки. Это будет особенно непросто, потому что все будут гореть желанием взяться за работу. Однако из-за огромного размера проекта без критической фазы планирования и структурирования вас поджидает неудача.

Борьба с законом Конвея

В 1968 году Мелвин Конвей[1] (Melvin Conway) сформулировал правило, которое позднее получило название закона Конвея[2]: «Организации, проектирующие системы, ограничены дизайном, который копирует структуру коммуникаций в этой организации». По мнению Конвея, централизованная организация с вертикальной иерархией может порождать только централизованные вертикальные архитектуры и никогда не породит распределенную архитектуру. Аналогичным образом организация, структурированная по функциональным границам, будет производить только функциональные декомпозиции систем. Безусловно, в эпоху цифровых коммуникаций закон Конвея не универсален, и все же его проявления встречаются достаточно часто.

Если закон Конвея создает угрозу для успеха вашего проекта, хорошим практическим способом противодействия ему станет изменение структуры вашей организации. Для этого необходимо сначала выработать правильную, адекватную архитектуру, а затем отразить ее в организационной структуре организации, структуре отчетности и линиях коммуникаций. Не стесняйтесь предлагать такую реорганизацию в составе своих рекомендаций по проектированию на анализе SDP.

Хотя Конвей изначально имел в виду проектирование систем, его закон в равной степени относится к планированию проекта и природе сети. Если план

---

[1] Melvin E. Conway, «How Do Committees Invent?», *Datamation*, 14, no. 5 (1968): 28–31.

[2] https://ru.wikipedia.org/wiki/Закон_Конвея

проекта включает сеть сетей, возможно, вам придется сопроводить его планом реструктурирования организации, который повторяет эти сети. Степень, до которой вам придется противодействовать закону Конвея, даже в проекте обычного размера сильно зависит от ситуации. Учитывайте организационную динамику и разработайте правильную структуру, если ваши наблюдения (или даже ваша интуиция) подсказывают вам, что это необходимо.

Малые проекты

На другом конце оси масштаба находятся малые (и даже очень малые) проекты. Как ни странно, даже такие малые проекты тоже необходимо тщательно планировать. Малые проекты еще сильнее подвержены ошибкам планирования, чем проекты нормального размера. Из-за своего размера они гораздо сильнее реагируют на изменения в условиях. Представьте эффект неправильного назначения человека из команды. В команде из 15 человек такая ошибка повлияет приблизительно на 7% доступных ресурсов. В команде из 5 человек она повлияет на 20% ресурсов проекта. Возможно, ошибку в 7% проект еще переживет, но ошибка в 20% указывает на серьезные неприятности. Большой проект может содержать буфер ресурсов, который поможет пережить ошибку. В малых проектах каждая ошибка критична.

С другой стороны, малые проекты могут быть настолько простыми, что они не требуют особого планирования проекта. Например, если в проекте используется всего один ресурс, сеть проекта представляет собой длинную цепочку активностей, продолжительность которой равна сумме продолжительностей по всем активностям. При таких минималистских планах проекта вы знаете продолжительность и затраты. В построении кривой «время-затраты» или вычислении риска нет необходимости (риск всегда равен 1,0). Так как в большинстве проектов присутствует некоторая разновидность сети, которая отличается от простой цепочки, а вам всегда следует избегать субкритических проектов, в практическом смысле даже простые проекты всегда необходимо планировать.

Планирование по уровням

Все примеры планирования проектов, встречавшиеся в книге, строили свою сеть активностей на основании логических зависимостей между активностями. Я называю такой подход *планированием по зависимостям*. Тем не менее существует и другой вариант, а именно построение проекта в соответствии с его архитектурными уровнями. При использовании архитектурной структуры Метода этот процесс получается весьма прямолинейным. Вы можете начать с построения *Вспомогательных средств*, перейти к *Ресурсам* и компо-

нентам *Доступ к ресурсу*, а затем заняться *Ядрами*, *Менеджерами* и *Клиентами*, как показано на рис. 12.11. Я называю этот метод *планированием по уровням*.

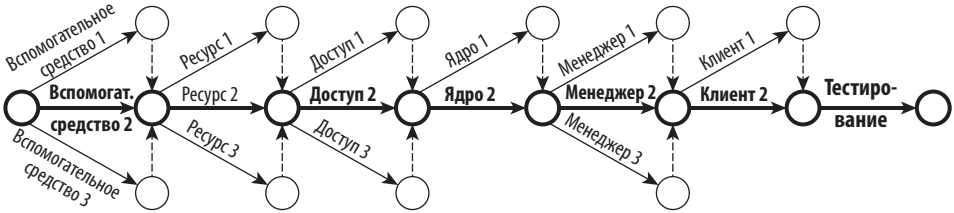


Рис. 12.11. Планирование проекта по уровням

Как следует из рис. 12.11, диаграмма сети фактически представляет собой серию импульсов, каждый из которых соответствует уровню архитектуры. Хотя импульсы последовательны, во внутренней реализации каждый импульс строится в параллельном режиме. Соблюдение Методом принципа закрытой архитектуры обеспечивает возможность параллельной работы в импульсе.

При планировании по уровням график получается сходным с графиком того же проекта, спланированного по зависимостям. В обоих случаях формируется критический путь, состоящий из компонентов архитектуры на разных уровнях.

> **ВНИМАНИЕ** Занимаясь планированием по уровням, не забудьте добавить в сеть неструктурные активности, например явную интеграцию и тестирование системы.

Достоинства и недостатки

Оборотной стороной планирования по уровням является возрастание рисков. Теоретически если все сервисы каждого уровня имеют одинаковую продолжительность, то все они являются критическими, а показатель риска приближается к 1,0. Впрочем, даже если это не так, любая задержка в завершении любого уровня немедленно задерживает весь проект, потому что следующие импульсы приостанавливаются. Однако при планировании по зависимостям только критические активности создают риск задержки проекта. Лучший (и почти обязательный) способ решения проблемы высокого риска в проектах с планированием по уровням — разуплотнение риска. Так как почти все активности будут критическими или околокритическими, проект будет очень хорошо реагировать на разуплотнение, так как все активности в каждом импульсе получают дополнительный временной резерв. Чтобы дополнительно компенсировать неявный риск планирования по уровням, проект следует разуплотнить, чтобы

риск был менее 0,5 — возможно, до 0,4. Этот уровень разуплотнения предполагает, что проекты, спланированные по уровням, будут занимать больше времени, чем проекты, спланированные по зависимостям.

Планирование по уровням может увеличить размер команды, что, в свою очередь, приведет к возрастанию прямых затрат проекта. При планировании по зависимостям вы ищете минимальный уровень ресурсов, обеспечивающий беспрепятственное перемещение по критическому пути за счет обмена временного резерва на ресурсы. При планировании по уровням вам может понадобиться столько ресурсов, сколько необходимо для завершения текущего уровня. Команде придется работать параллельно над всеми активностями внутри каждого импульса и завершить их все перед тем, как переходить к следующему импульсу. Вы должны предполагать, что все компоненты текущего уровня необходимы для следующего уровня.

С учетом сказанного планирование по уровням обладает очевидным преимуществом: оно дает очень простой план проекта для исполнения. Это лучшее противоядие для сложных сетей проектов, позволяющее сократить общую цикломатическую сложность вдвое и более. Теоретически, поскольку все импульсы идут последовательно, в любой момент времени менеджеру проекта приходится бороться только со сложностью исполнения каждого импульса и вспомогательных активностей. Цикломатическая сложность каждого импульса приблизительно равна количеству параллельных активностей. В типичной системе на базе Метода эта цикломатическая сложность равна всего 4 или 5, тогда как цикломатическая сложность проектов, спланированных по зависимостям, может достигать 50 и более.

Многие проекты в программной отрасли справляются как с отставанием от графика, так и с избытком мощностей; следовательно, настоящие проблемы создает сложность, а не продолжительность и не затраты. Там, где это возможно, в системах на базе Метода я предпочитаю применять планирование по уровням для решения проблем рискованного и сложного исполнения. Как обычно при планировании проектов, для планирования по уровням необходимо заранее иметь правильную архитектуру.

Методы планирования по уровням и планирования по зависимостям могут объединяться. Например, в примере из главы 11 все *Вспомогательные средства* инфраструктуры были перемещены в начало проекта, несмотря на тот факт, что их логические зависимости позволили бы выполнить их на гораздо более поздней стадии проекта. Оставшаяся часть проекта планировалась на основании логических зависимостей.

> **ПРИМЕЧАНИЕ** У планирования по уровням и планирования по зависимостям различаются только методологии планирования исходных сетевых зависимостей. Все остальные методы планирования проектов, рассмотренные в предыдущих главах, применяются точно так же.

Уровни и построение

Планирование и построение по уровням — отличный пример правила проектирования из главы 4: *функции всегда и везде являются аспектами интеграции, а не реализации*. Только после того, как все уровни будут завершены, их можно будет интегрировать в функции. Отсюда следует, что планирование по уровням лучше подходит для обычных проектов, нежели для более крупных и более сложных проектов с несколькими независимыми подсистемами. Возвращаясь к аналогии с домом, простые дома почти всегда строятся по уровням: фундамент, коммунальные сети, стены, крыша и т. д. В большом многоэтажном здании каждый этаж представляет собой самостоятельный проект с собственными коммунальными сетями, стенами, потолком и пр.

Напоследок заметим, что планирование проекта по уровням фактически означает разбиение проекта на меньшие подпроекты. Эти меньшие проекты реализуются последовательно и разделяются временными точками сопряжения. Такой подход имеет много общего с разбиением мегапроекта на меньшие сети и обладает похожими преимуществами.

13 Пример планирования проекта

Хотя в главе 11 был представлен пример проекта, я прежде всего хотел продемонстрировать процесс мышления при применении методов планирования проекта и связи между ними. Демонстрация планирования проекта на всех стадиях была вторичной целью. В этой главе основное внимание уделяется управлению принятием решений по планированию в реальных проектах, а также тому, когда следует применять те или иные методы планирования. Проект, планированием которого мы займемся, строит систему TradeMe из главы 5. Как и в примере проектирования системы из главы 5, эта глава строится непосредственно на материале реального проекта, разработанного компанией IDesign для одного из своих заказчиков. Рабочая группа состояла из двух архитекторов IDesign (эксперта и ученика) и менеджера проекта со стороны заказчика. При том, что конкретные бизнес-подробности в этом примере удалены или замаскированы, я представляю процесс планирования проекта в его реальном виде. Как работа по проектированию системы, так и планирование проекта были завершены менее чем за неделю.

Все данные и вычисления, использованные в этой главе, доступны в загружаемых файлах. Тем не менее, когда вы будете читать эту главу впервые, я рекомендую бороться с искушением постоянно переключаться между текстом и файлами. Вместо этого лучше сосредоточиться на рассуждениях, ведущих к этим вычислениям, и интерпретации их результатов. А когда вы начнете понимать логику происходящего, используйте эту главу как справочник для более подробного исследования данных, чтобы подтвердить свое понимание и потренироваться в практическом применении этих приемов.

> **ВНИМАНИЕ** Эта глава не повторяет предыдущие главы и в ней не объясняются конкретные приемы планирования проектов. Более полное понимание материала предыдущих глав поможет вам извлечь максимум пользы из этого примера.

Оценки

В ходе работы по планированию проекта TradeMe применялись оценки двух видов: оценки отдельных активностей и общая оценка проекта. Оценки отдельных активностей использовались в решениях по планированию проекта, а общая оценка — для проверки результатов планирования.

Оценки отдельных активностей

Оценка отдельных активностей началась с составления списка активностей в проекте, чтобы случайно не упустить какие-нибудь критические активности. Команда разделила активности TradeMe на три категории:

- Структурные активности, связанные с программированием.
- Неструктурные активности, связанные с программированием.
- Активности, не связанные с программированием.

При построении списка активностей команда проектирования расширила каждый список, включив в него отдельные активности и оценки продолжительности для каждой активности. Команда также обозначила роль, ответственную за каждую активность, в соответствии с технологическими процессами заказчика или своим собственным опытом.

Предположения оценки

Команда проектирования четко документировала все исходные ограничения и предположения по своим оценкам. Проект TradeMe основывался на следующих предположениях:

- *Подробное проектирование*. Отдельные разработчики могли выполнить подробное проектирование, так что у каждой активности программирования была собственная фаза подробного проектирования.
- *Процесс разработки*. Команда была настроена на быстрое и чистое построение системы с применением многих передовых методов, представленных в книге.

Структурные активности

Структурные активности TradeMe напрямую определялись системной архитектурой (см. рис. 5.14). К их числу принадлежали *Вспомогательные средства*, *Ресурсы*, *Доступ к ресурсам*, *Менеджеры*, *Ядра* и *Клиенты*; в основном это были задачи, предназначенные для разработчиков. Архитектор отвечал за ключевые

активности *Шины сообщений* и *Репозитория потоков операций*. В табл. 13.1 приведены оценки продолжительности для некоторых структурных активностей, связанных с программированием.

Таблица 13.1. Оценка продолжительности для некоторых структурных активностей, связанных с программированием

Идентификатор	Активность	Продолжительность (дни)	Роль
14	Журнал	10	Разработчик
15	Шина сообщений	15	Архитектор
16	Безопасность	20	Разработчик
18	База данных платежей	5	Архитектор базы данных
...	...	...	...
23	Репозиторий потоков операций	15	Архитектор
...	...	...	...
26	Доступ к платежам	10	Разработчик
...	...	...	...
35	Ядро поиска	15	Разработчик
...	...	...	...
38	Менеджер рынка	10	Разработчик
...	...	...	...
45	Административное приложение	25	Разработчик

Неструктурные активности

Команда проектирования TradeMe выявила ряд активностей, связанных с программированием, которые не имели прямого соответствия в архитектуре. Эти активности были результатом как оперативных концепций системы, так и процессов разработки, принятых в компании. В табл. 13.2 приведены оценки продолжительности неструктурных активностей, связанных с программированием.

Таблица 13.2. Оценка продолжительности для неструктурных активностей, связанных с программированием

Идентификатор	Активность	Продолжительность (дни)	Роль
10	Оснастка для тестирования системы	25	Инженер по тестированию
36	Абстрактный менеджер	30	Разработчик
40	Оснастка для регрессионного тестирования	10	Разработчик

Абстрактный *Менеджер* является базовым сервисом для остальных *Менеджеров* в системе. Он содержит основную часть управления потоком операций, а также реализацию взаимодействий с шиной сообщений. Производные *Менеджеры* выполняют конкретные потоки операций. Обе другие активности были связаны с тестированием. Оснастка для тестирования системы была закреплена за инженером по тестированию, но оснастка для регрессионного тестирования принадлежала разработчику.

Активности, не связанные с программированием

Проект TradeMe содержит много активностей, не связанных с программированием, которые обычно концентрируются в начале или в конце проекта. Активности, не связанные с программированием, принадлежали разным участникам основной команды: инженеру по тестированию, тестировщикам или внешним экспертам (скажем, UX-проектировщику). Эти активности показаны в табл. 13.3. На содержимое списка также повлиял процесс разработки, принятый в компании, предпосылки планирования и обязательства по качеству.

Таблица 13.3. Оценка продолжительности для активностей, не связанных с программированием

Идентификатор	Активность	Продолжительность (дни)	Роль
2	Требования	15	Архитектор, менеджер продукта
3	Архитектура	15	Архитектор, менеджер продукта
4	План проекта	10	Архитектор, менеджер проекта, менеджер продукта

Таблица 13.3 (*окончание*)

Идентификатор	Активность	Продолжительность (дни)	Роль
5	Общение с руководством	5	Архитектор, менеджер проекта, менеджер продукта
7	UX-проектирование	10	Эксперт в области UI/UX
8	Обучение разработчиков	5	Архитектор
9	План тестирования	25	Инженер по тестированию
11	Построение и настройка	10	DevOps
12	UI-проектирование	20	Эксперт в области UI/UX
13	Документация	10	Менеджер продукта
25	Миграция данных	10	Разработчик
46	Доработка документации	10	Менеджер продукта
47	Тестирование системы	10	Контроль качества
48	Выпуск системы	10	Архитектор, менеджер проекта, менеджер продукта, DevOps

Общая оценка проекта

Группа проектирования предложила группе из 20 человек дать оценку проекта TradeMe в целом. Единственными входными данными была статическая архитектура TradeMe и оперативная концепция системы. Команда воспользовалась методом широкополосной оценки и получила продолжительность в 10,5 месяца при среднем комплектовании 7,1 человека. Таким образом, общие затраты составили 74,6 человеко-месяца.

Зависимости и сеть проекта

Затем команда проектирования перешла к определению зависимостей между различными активностями. Отправной точкой для системы TradeMe была архитектура и зависимости в поведении между структурными компонентами. Для них команда добавила зависимости, не связанные с поведением (такие,

как активности, не связанные с программированием, или активности программирования, не зависящие от архитектуры). Команда проектирования также применила паттерны планирования проектов и методы сокращения сложности для упрощения сети и предстоящего исполнения проекта. Результатом стала первая итерация сети проекта.

Поведенческие зависимости

При построении первой группы зависимостей команда проектирования проанализировала сценарии использования и цепочки вызовов, которые их поддерживают. Для каждой цепочки вызовов были перечислены все компоненты в цепочке (часто в порядке архитектурной иерархии — например, *Ресурсы* сначала, *Клиенты* в последнюю очередь), а затем добавлены зависимости. Например, при анализе сценария использования Добавление мастера (см. рис. 5.18) команда проектирования заметила, что *Менеджер принадлежности* обращается с вызовом к *Ядру нормативов*, поэтому *Ядро нормативов* было добавлено как предшественник *Менеджера принадлежности*.

Извлечение зависимостей из сценариев использования выполнялось за несколько проходов, потому что каждая цепочка зависимостей теоретически открывала разные зависимости. Команда проектирования даже обнаружила в цепочках вызовов некоторые отсутствующие зависимости. Например, на основании исключительно цепочек вызовов из главы 5 *Ядро нормативов* требовало только сервиса *Доступ к нормативам*. При дальнейшем анализе группа проектирования решила, что *Ядро нормативов* также зависит от компонентов *Доступ к проектам* и *Доступ к подрядчикам*.

Абстрактные структурные зависимости

Абстрактный Менеджер инкапсулировал общие действия управления потоком операций (например, хранение данных или управление состоянием). Команда проектирования добавила зависимость между *Абстрактным менеджером* и *Репозиторием потоков операций*. Другие *Менеджеры* зависели от *Абстрактного Менеджера*. Аналогичным образом *Абстрактный Менеджер* предоставил зависимость от *Шины сообщений* для всех *Менеджеров*.

Оперативные зависимости

Некоторые зависимости неявно присутствовали в цепочках вызовов вследствие оперативной концепции системы. В TradeMe все взаимодействия между *Клиентами* и *Менеджерами* (а также между *Менеджерами* и другими *Менеджерами*) проходили через шину сообщений, что создавало оперативные (а не структурные) зависимости между ними. Зависимости показали, что готовность *Менеджеров* была необходима *Клиентам* для тестирования и развертывания.

Зависимости, не связанные с поведением

Система TradeMe также содержит зависимости, которые невозможно было связать напрямую с требуемым поведением системы или ее оперативной концепцией. В их число входили активности как связанные, так и не связанные с программированием. Такие зависимости в основном были обусловлены процессом разработки, принятым в компании, и предпосылками планирования TradeMe. Например, новая система должна была переносить унаследованные данные из старой системы. Миграция данных требует, чтобы новые *Ресурсы* (базы данных) завершались первыми, так что активность миграции данных зависит от *Ресурсов*. Аналогичным образом для завершения *Менеджеров* требовалась *Оснастка для регрессионного тестирования*. Кроме того, во время планирования проекта план должен принимать во внимание несколько оставшихся активностей начальной стадии. Наконец, компания имеет собственные процедуры выпуска и внутренние зависимости, которые были воплощены в виде зависимостей между завершающимися активностями.

Переопределение некоторых зависимостей

В TradeMe основной оперативной концепцией было использование шины сообщений. Было очень важно правильно выбрать подходящую технологию шины сообщений и совместить подробное планирование и активности сообщений и контрактов с шиной сообщений. Зависимости, обусловленные цепочками вызовов, показали, что проект может отложить активность *Шина сообщений* до того момента, когда она потребуется *Клиентам* и *Менеджерам*. Тем не менее при этом появлялся риск того, что шина сообщений, выбранная командой разработки, может свести на нет предыдущие решения относительно проектирования или реализации. Команда решила, что будет безопаснее разобраться с активностью *Шины сообщений* в самом начале проекта.

Аналогичная логика была применена к безопасности. Хотя анализ цепочки вызовов показывал, что явные действия безопасности должны были предприниматься *Клиентами* и *Менеджерами*, безопасность была настолько важна, что проект должен был обеспечить завершение *Безопасности* перед всеми активностями бизнес-логики. Это гарантировало, что все активности будут поддерживать средства безопасности, если она им потребуется, и позволяло избежать проблем из-за попыток реализовать поддержку безопасности «задним числом».

Сокращение сложности

Команда проектирования также переопределила зависимости для сокращения сложности формируемой сети. А именно были изменены следующие зависимости:

- **Реализация инфраструктуры в начале**. В TradeMe большинство активностей зависело от *Вспомогательных средств*, таких как *Журнал*. Перемещение инфраструктуры (которая также включала *Сборку*) в начало проекта привело к радикальному сокращению зависимостей в проекте. У него также было еще одно преимущество: инфраструктура становилась доступной для всех компонентов в случае необходимости (особенно тех, у которых не было очевидной необходимости на основании одних цепочек вызовов).
- **Добавление контрольных точек**. Даже на этой ранней стадии проекта команда проектирования добавила три контрольные точки. Контрольная точка *Анализ SDP* завершает активности начальной стадии. Две другие контрольные точки — *Завершение инфраструктуры* и *Завершение Менеджеров*: все активности разработки зависели от инфраструктурной контрольной точки, а все *Клиенты* зависели от завершения *Менеджеров*.
- **Консолидация унаследованных зависимостей**. Команда проектирования консолидировала зависимости в унаследованные зависимости там, где это было возможно. Например, хотя *Клиенты* требовали *Шины сообщений*, эта зависимость может быть унаследована через их зависимости от *Менеджеров*.

Проверки на здравый смысл

После того как исходная структура сети была определена, команда проектирования выполнила ряд проверок на здравый смысл:

1. Проект TradeMe имеет одну стартовую активность и одну конечную активность.
2. Каждая активность в проекте находится на пути, который завершается где-то на критическом пути(-ях).
3. Исходное вычисление риска дает относительно низкий показатель риска.
4. Вычисление продолжительности проекта без назначения ресурсов. Полученный результат — 7,8 месяца — позднее будет использован для важной проверки нормального решения.

Нормальное решение

Компания предоставила следующие предположения планирования:

- *Основная команда*. Основная команда требовалась на протяжении всего проекта. Она состояла из одного архитектора, менеджера проекта и менеджера продукта. Возможность напрямую работать над проектом предоставлялась основной команде относительно редко. Такая работа включала ключевые активности с высоким риском, которые выполнялись архитектором, а также подготовку документации пользователя, порученную менеджеру продукта.

- *Доступность экспертов*. Проекту доступны эксперты или специалисты: инженеры по тестированию, архитекторы баз данных и проектировщики UX/UI.
- *Назначение*. Между разработчиками и сервисами (или другими активностями, связанными с программированием) существует однозначное соответствие. Помимо назначения разработчиков на основании временного резерва, система TradeMe по возможности поддерживала непрерывность задач (см. главу 7).
- *Контроль качества*. Один тестировщик должен был участвовать в проекте от начала построения до конца проекта. Он рассматривался как прямые затраты только в ходе активности тестирования системы. Один дополнительный тестировщик требовался для активности тестирования системы.
- *Построение и операции*. Один специалист по построению, настройке конфигурации, развертыванию и DevOps должен был участвовать в проекте от начала построения до конца проекта.
- *Разработчики*. Разработчики между задачами рассматривались как прямые, а не косвенные затраты. Высокие стандарты качества TradeMe сняли необходимость в разработчиках во время тестирования системы.

В табл. 13.4 описаны роли, требуемые в каждой фазе проекта.

Таблица 13.4. Роли и фазы проекта

Роль	Начальная стадия	Инфраструктура	Сервисы	Тестирование
Архитектор	X	X	X	X
Менеджер проекта	X	X	X	X
Менеджер продукта	X	X	X	X
Тестировщики		X	X	X
DevOps		X	X	X
Разработчики		X	X	

Диаграмма сети

Назначение ресурсов на различные активности повлияло на сеть проекта. В нескольких местах в сети появились зависимости от ресурсов помимо логических зависимостей между активностями. После консолидации унаследованных зависимостей диаграмма сети выглядела так, как показано на рис. 13.1.

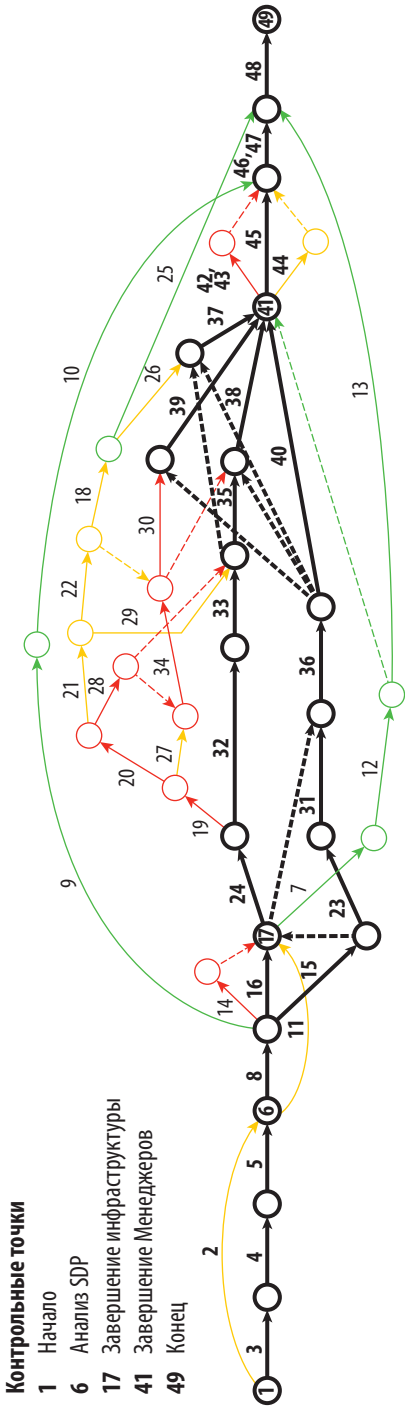


Рис. 13.1. Диаграмма сети логических зависимостей

Рисунок 13.1 содержит пару свернутых зависимостей (обозначенных двумя номерами активностей на стрелку), которая упрощает диаграмму без ее изменения. Самая замечательная особенность этой диаграммы сети — наличие двух критических путей.

Планируемый прогресс

На рис. 13.2 изображена планируемая осваиваемая ценность первого нормального решения. Продолжительность этого решения составляла 7,8 месяца, что указывает на то, что распределение комплектования не привело к расширению критического пути. Диаграмма на рис. 13.2 своей формой в целом напоминает пологую S-образную кривую, но соответствие не идеально. Проект начинается относительно неплохо, но вторая половина проекта не выглядит пологой. Крутая кривая планируемой осваиваемой ценности также отразилась в повышенных значениях риска. И риск активности, и риск критичности были равны 0,7.

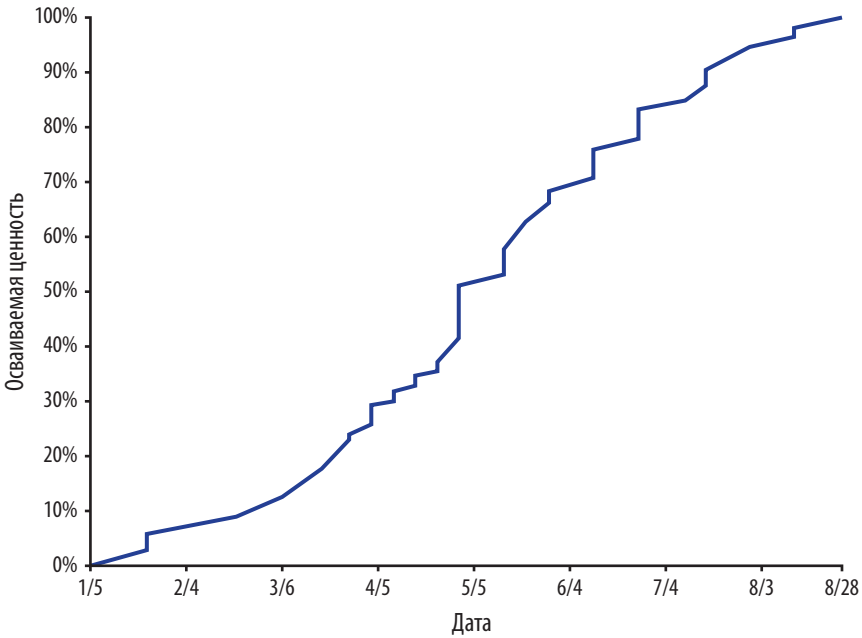


Рис. 13.2. Планируемая осваиваемая ценность первого нормального решения

Планируемое распределение комплектования

На рис. 13.3 изображена диаграмма распределения комплектования первого нормального решения. Как и в случае с диаграммой планируемой осваиваемой

ценности, распределение на рис. 13.3 указывает на наличие проблем. Явный пик в центре проекта свидетельствует о неэффективном расходовании ресурсов и нереалистичных ожиданиях относительно эластичности комплектования (см. главу 7 и рис. 7.10).

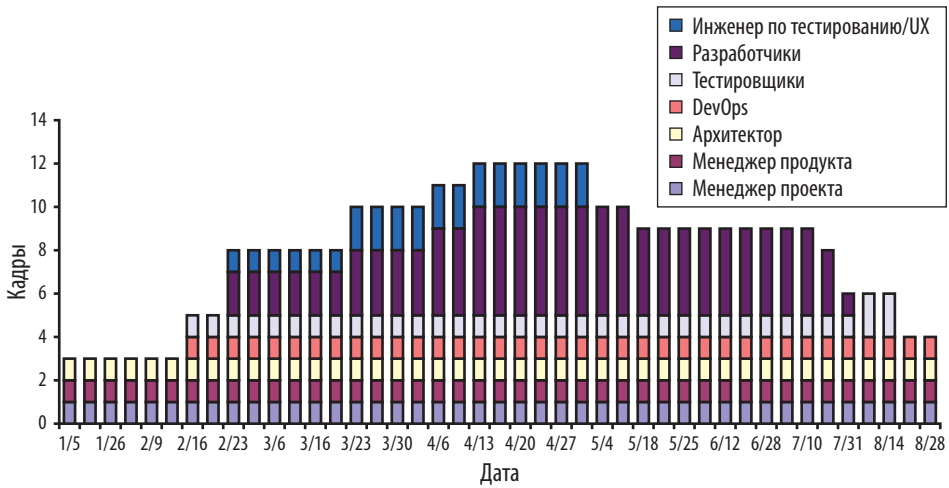


Рис. 13.3. Диаграмма распределения комплектования первого нормального решения

Затраты и эффективность

На основании распределения комплектования общие затраты проекта составили 59 человеко-месяцев: 32 человеко-месяца прямых затрат и 27 человеко-месяцев косвенных затрат. Более высокие прямые затраты по сравнению с косвенными указывали на то, что это решение с большой вероятностью расположено в левой части кривой «время-затраты», где косвенные затраты все еще низки.

Вычисленная эффективность проекта была равна 32%. Поскольку верхний практический предел составлял 25%, такая высокая эффективность была спорной. Сочетание всех этих факторов — прямые затраты выше косвенных, четко выраженный пик на диаграмме распределения комплектования, высокая эффективность — убедительно указывало на излишне агрессивные предположения относительно эластичности комплектования.

Решение ожидает, что по всем параллельным сетевым путям ресурсы всегда будут доступны в нужный момент для продвижения проекта. Крутая диаграмма планируемой осваиваемой ценности наглядно представляет это ожидание. Короче говоря, первая попытка построения нормального решения предполага-

ет исключительно высокую эффективность команды — скорее всего, слишком высокую, чтобы быть реальной.

Сводка результатов

В табл. 13.5 приведена сводка метрик проекта первого нормального решения.

Таблица 13.5. Метрики проекта для первого нормального решения

Метрика проекта	Значение
Продолжительность (месяцы)	7,8
Общие затраты (человеко-месяцы)	59
Прямые затраты (человеко-месяцы)	32
Пиковое комплектование	12
Среднее комплектование	7,5
Средняя численность разработчиков	3,5
Эффективность	32%
Риск активности	0,7
Риск возникновения критичности	0,7

Уплотненное решение

На следующем шаге рассматривались варианты ускорения проекта. Из-за наличия двух критических путей оптимальным планом действий было уплотнение проекта за счет введения параллельной работы.

Из рис. 13.1 очевидно, что сервисы *Менеджер* (активности 36, 37, 38, 39) вместе с *Оснасткой регрессионного тестирования* (активность 40) завершали два критических пути, а также два околокритических пути. В свою очередь, *Клиенты* (активности 42, 43, 44, 45) зависели от завершения всех *Менеджеров*, что затягивало проект. Таким образом, *Клиенты* и *Менеджеры* были естественными кандидатами для сжатия.

Добавление вспомогательных активностей

Для каждого сервиса *Менеджер* команда проектирования добавила следующие активности, которые делали возможным уплотнение:

1. Активность проектирования контракта, отделявшая *Клиентов* от *Менеджера*. Возможно, различные активности проектирования контракта могли стартовать после анализа SDP, но было решено, что их лучше отложить до завершения инфраструктуры. Оценка объема работы на контракт: 5 дней.
2. Имитатор *Менеджера*, который предоставлял достаточно хорошую реализацию контракта *Менеджера*. Задача имитаторов — сделать возможной полноценную разработку *Клиентов*, которые теперь зависели от имитаторов, а не от реальных *Менеджеров*. Имитаторы не зависели от сервисов более низкого уровня (таких, как *Доступ к ресурсам* или *Ядро*). Для создания имитаторов были необходимы только контракты *Менеджеров* и *Шина сообщений*. Сами контракты зависели от инфраструктуры, которая включала *Шину сообщений*. Оценка объема работы на имитатор: 15 дней.
3. Специальная активность для интеграции *Клиентов* с *Менеджерами* и их повторного тестирования. Активность интеграции зависела от завершения реального *Менеджера* и его *Клиентов*. Активность тестирования системы теперь требовала завершения не только *Клиентов*, но и всех интеграций *Менеджера*. Оценка объема работы на активность интеграции: 5 дней.

На рис. 13.4 изображена упрощенная диаграмма сети; активности, связанные с уплотнением, выделены красным цветом. В уплотненной сети *Менеджеры* были только околокритическими и разрабатывались приблизительно по тому же графику, что и нормальное решение. Самое важное изменение (которое сделало возможным само уплотнение) заключалось в том, что *Клиенты* теперь завершались на месяц быстрее. При этом сокращение продолжительности проекта составляло меньше месяца из-за дополнительных активностей интеграции после завершения *Менеджеров*.

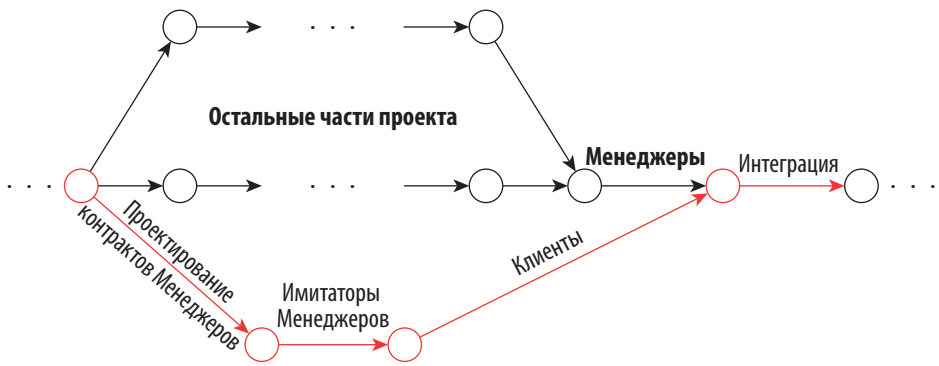


Рис. 13.4. Упрощенная диаграмма сети для уплотненного решения

Оценка продолжительности для Менеджеров

Оценка продолжительности для *Менеджеров* осталась неизменной. В нормальном решении каждая активность *Менеджера* должна выполнить некоторую внутреннюю работу по проектированию контракта сервиса. Теоретически после того, как команда проектирования выделила проектирование контракта из *Менеджеров* в отдельные активности, каждый *Менеджер* должен занять меньше времени. Тем не менее на практике такое сокращение маловероятно. Расщепление активностей никогда не обладает 100-процентной эффективностью, и часть усилий неизбежно тратится на необходимость разобраться в контракте и его влиянии на внутреннюю реализацию *Менеджеров*. Для компенсации этих недостатков команда проектирования сохранила для продолжительности *Менеджеров* такую же оценку, как у обычного решения.

Назначение ресурсов

Остальные этапы уплотненного решения были практически идентичны этапам нормального решения. Однако команда проектирования обнаружила, что персонал можно сократить на двух разработчиков, используя архитектора для одной активности разработки и сместив график на одну неделю. Компания решила, что небольшая задержка ради сокращения численности приемлема с учетом трудностей с привлечением дополнительных разработчиков. Продолжительность уплотненного решения составила 7,1 месяца, ускорение — на 3 недели (9%) по сравнению с нормальным решением (7,8 месяца). Новые ресурсы потребляли большее количество временного резерва, а новый показатель риска для проекта был равен 0,74.

Планируемый прогресс

На рис. 13.5 представлена диаграмма планируемой осваиваемой ценности для уплотненного решения. Кривая несколько понижается в конце проекта — лучше, чем у нормального решения.

Планируемое распределение комплектования

На рис. 13.6 изображена диаграмма распределения комплектования уплотненного решения. Диаграмма выглядит в целом нормальной. Исходный рост с 3 до 12 человек создает некоторые проблемы, но не смертелен. Пиковое комплектование 12 — такое же, как у нормального решения. Среднее комплектование равно 8,2 (по сравнению с 7,5 у нормального решения).

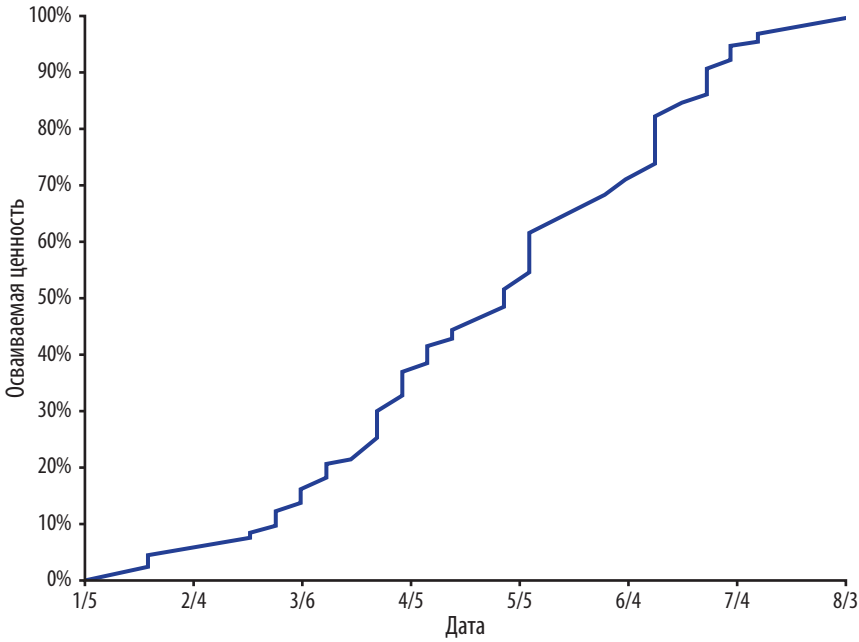


Рис. 13.5. Планируемая осваиваемая ценность уплотненного решения

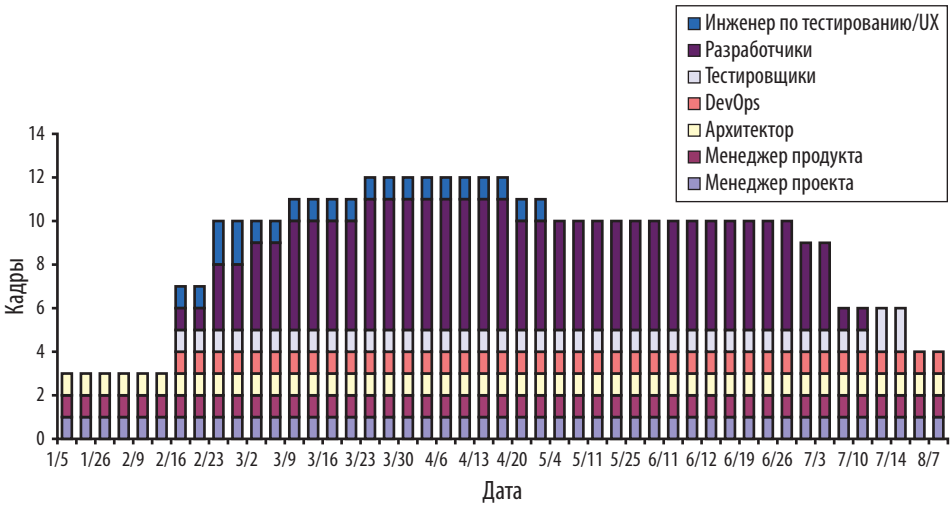


Рис. 13.6. Диаграмма распределения кадров уплотненного решения

Затраты и эффективность

Затраты уплотненного решения составили 58,5 человеко-месяца — чуть меньше 59 человеко-месяцев у нормального решения. Прямые затраты составили 36,7 человеко-месяца по сравнению с 32 человеко-месяцами у нормального решения. И хотя это решение реализовывалось быстрее и требовало меньших затрат, настоящее отличие от нормального решения проявлялось в ожидаемой эффективности проекта — 37%. Если эффективность 32% нормального решения требовала чрезвычайно эффективной работы команды, для уплотненного решения подошла бы разве что команда героев. В сочетании с повышенным риском 0,74 уплотненное решение было верным путем к катастрофе.

Сводка результатов

В табл. 13.6 приведена сводка метрик уплотненного решения. Уплотненное решение сделало уже непростой проект (см. рис. 13.1) еще более сложным и создало нереалистично высокие ожидания к эффективности. Впрочем, главной проблемой была интеграция, а не возрастание сложности исполнения. Множественные параллельные интеграции, происходящие в конце проекта, не оставляли свободы для маневра. Если что-то в них пойдет не так, у команды не останется времени для исправлений. Рост сложности исполнения и риск интеграции не компенсировался месяцем уплотнения.

Таблица 13.6. Метрики проекта для уплотненного решения

Метрика проекта	Значение
Продолжительность (месяцы)	7,1
Общие затраты (человеко-месяцы)	58,5
Прямые затраты (человеко-месяцы)	36,7
Пиковое комплектование	12
Среднее комплектование	8,2
Средняя численность разработчиков	4,7
Эффективность	37%
Риск активности	0,73
Риск возникновения критичности	0,75

Даже при этом попытка уплотнения не была напрасной тратой времени — она доказала, что уплотненное решение было обречено на неудачу. Уплотненное решение также помогло группе проектирования лучше понять проект и предоставило другую точку на кривой «время-затраты».

Планирование по уровням

Главной проблемой первого нормального решения была не нереалистичная эффективность, а сложность сети проекта. Чтобы эта сложность стала очевидной, достаточно проанализировать (уже упрощенную) диаграмму сети на рис. 13.1. Цикломатическая сложность сети равна 33 единицам. В сочетании с высокой эффективностью, ожидаемой от команды, это создавало более высокий риск исполнения.

Вместо того чтобы бороться с высокой сложностью, команда проектирования решила перепланировать проект по уровням архитектуры (вместо логических зависимостей между активностями). Результат применения этого подхода породил в основном цепочку импульсов активностей. Импульсы соответствовали уровням архитектуры или фазам проекта: начальная стадия, инфраструктура и подготовительная работа, *Ресурсы*, компоненты *Доступ к ресурсу*, *Ядра*, *Менеджеры*, *Клиенты* и активности выпуска (рис. 13.7).

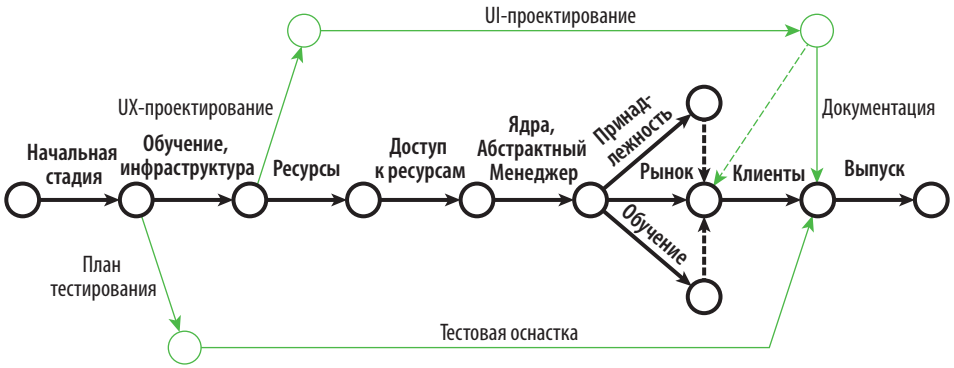


Рис. 13.7. Диаграмма сети при планировании по уровням

Хотя на диаграмме импульсы идут последовательно, во внутренней реализации они выполнялись параллельно. На рис. 13.7 все импульсы свернуты, кроме расширенного импульса *Менеджера*. Оставшиеся вспомогательные активности, такие как *UI-проектирование* и *Тестовая оснастка*, не были частью цепочки импульсов, но имели очень высокий временной резерв.

Из рис. 13.7 сразу видно, насколько проста эта сеть по сравнению с рис. 13.1. Так как эти импульсы были последовательными во времени, менеджеру проекта оставалось только разбираться со сложностью каждого импульса и его вспомогательных активностей. В TradeMe сложности отдельных импульсов были равны 2, 4, 5, 4, 4, 4, 4 и 2. Сложность вспомогательных активностей была равна 1 ввиду их высокого временного резерва и практически не оказывала влияния на сложность исполнения.

Планирование по уровням и риск

Как обсуждалось в главе 12, планирование по уровням порождает более рискованные проекты. Команда проектирования обнаружила, что с TradeMe риск решения, спланированного по уровням, был равен 0,76 — более высокий по сравнению с 0,7 исходного нормального решения (которое использовало планирование по зависимостям). Если не обращать внимания на вспомогательные активности с высоким временным резервом, риск поднялся еще выше — до 0,79.

Распределение комплектования

На рис. 13.8 показано планируемое распределение комплектования для решения с планированием по уровням. Общая форма диаграммы выглядит вполне удовлетворительно. Проекту необходимы только 4 разработчика, а пиковое комплектование достигало 11 человек.

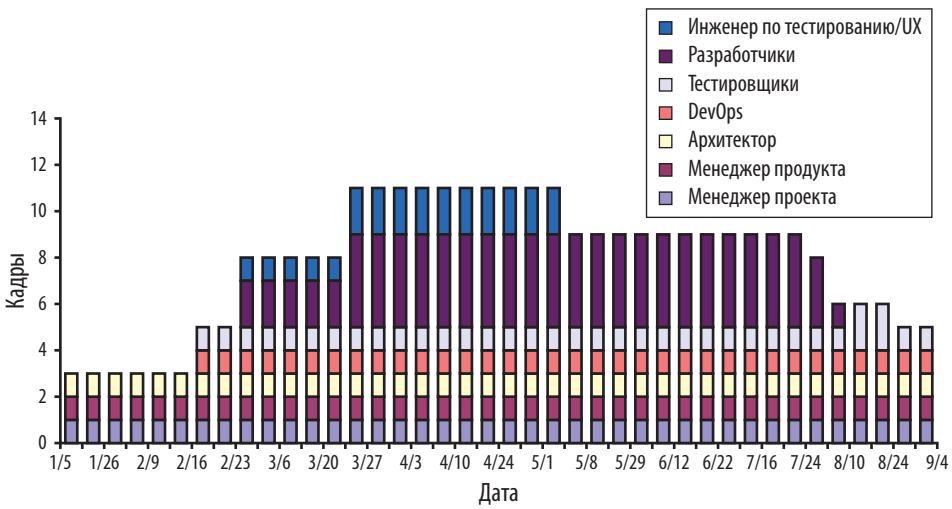


Рис. 13.8. Диаграмма распределения комплектования для планирования по уровням

Сводка результатов

В табл. 13.7 приведена сводка метрик для планирования TradeMe по уровням.

Таблица 13.7. Метрики проекта для планирования по уровням

Метрика проекта	Значение
Продолжительность (месяцы)	8,1
Общие затраты (человеко-месяцы)	60,8
Прямые затраты (человеко-месяцы)	32,2
Пиковое комплектование	11
Среднее комплектование	7,5
Средняя численность разработчиков	3,4
Эффективность	31%
Риск активности	0,75
Риск возникновения критичности	0,76

Субкритическое решение

Решение с планированием по уровням требовало четырех разработчиков. Компания беспокоилась о том, что могло произойти, если найти этих четырех разработчиков не удастся. Следовательно, было важно исследовать возможные последствия перехода в субкритическое состояние. Предпосылки планирования допускали обращение к внешним экспертам.

Для этого проекта любое решение с планированием по уровням, имеющее менее четырех разработчиков, становилось субкритическим, поэтому команда проектирования предложила исследовать решение с двумя разработчиками. Этим разработчикам также было поручено проектирование базы данных. Субкритическая диаграмма сети была сходна с изображенной на рис. 13.7, за исключением того, что во внутренней реализации каждый импульс состоял только из двух параллельных цепочек активностей.

Продолжительность, планируемый прогресс и риск

Субкритическое решение увеличило продолжительность проекта до 11,1 месяца. Кривая планируемой осваиваемой ценности (рис. 13.9) была почти прямой линией, а характеристика R^2 линии тренда линейной регрессии была равна 0,98.

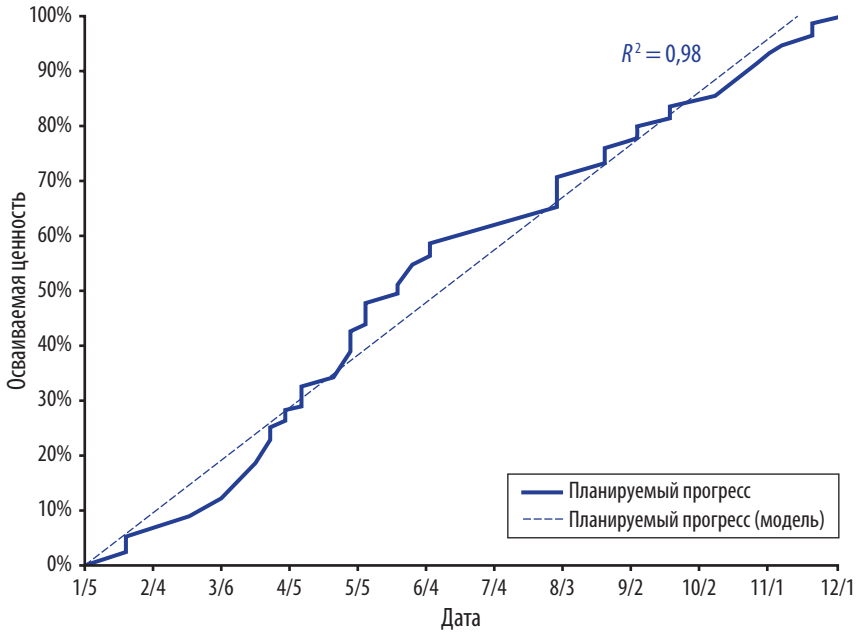


Рис. 13.9. Планируемый прогресс субкритического решения

Субкритическая природа решения также была отражена в его показателе риска 0,84. Если бы компании пришлось выбрать этот вариант, команда проектирования рекомендовала разуплотнить проект по крайней мере на месяц. В результате разуплотнения проект стал занимать около 12 месяцев — на 50% больше, чем в решении с планированием по уровням.

Затраты и эффективность

Общие затраты субкритического решения составили 74,1 человеко-месяца, при этом прямые затраты составили 30,4 человеко-месяца, а ожидаемая эффективность 25% выглядела более разумной. На диаграмме распределения комплектования (не приводится) отсутствовал «горб» в центре, что типично для субкритических решений (см. главу 7).

Сводка результатов

В табл. 13.8 приведены метрики проекта для субкритического решения.

Субкритические метрики времени и затрат (11,1 месяца и 74,1 человеко-месяца) выгодно отличались от аналогичных метрик общей оценки (10,5 месяца и 74,6 человеко-месяца), отличаясь приблизительно на 5% по продолжитель-

ности и менее 1% по затратам. Эта корреляция предполагала, что показатели субкритического решения были вероятным вариантом для проекта. Более реалистичная эффективность 25% также придавала достоверность субкритическому решению.

Таблица 13.8. Метрики проекта для субкритического решения

Метрика проекта	Значение
Продолжительность (месяцы)	11,1
Общие затраты (человеко-месяцы)	74,1
Прямые затраты (человеко-месяцы)	30,4
Пиковое комплектование	9
Среднее комплектование	6,7
Средняя численность разработчиков	2
Эффективность	25%
Риск активности	0,85
Риск возникновения критичности	0,82

Сравнение вариантов

Из анализа результатов табл. 13.5 и 13.7 можно сделать ряд содержательных выводов. Во-первых, продолжительность проекта оставалась в целом неизменной независимо от того, какой из методов применяла команда — планирование по уровням или планирование по зависимостям. Как объяснялось в главе 12, такое сходство было ожидаемым. В конце концов, зависимости на базе цепочки вызовов, по сути, являются производными от уровней, а продолжительность проекта определяется самым длинным путем по уровням. Кроме того, средний уровень комплектования разработчиков и эффективность не изменились. Основные отличия сводились к радикальному сокращению сложности исполнения и повышенному риску решения с планированием по уровням.

Короче говоря, для системы TradeMe планирование по уровням давало результаты, сравнимые с первым нормальным решением или превосходящие его во всех отношениях, кроме риска. Даже если решение с планированием по уровням обходилось дороже и занимало больше времени, простота его исполнения делала его очевидным кандидатом для TradeMe. Решение с планированием по уровням также было намного лучше субкритического решения, производного от него. Субкритическое решение требовало больших затрат, занимало больше времени и было более рискованным. Команда проектирования приняла реше-

ние с планированием по уровням как нормальное решение для оставшейся части этого анализа.

Планирование и риск

К этому моменту команда проектирования создала четыре решения для построения системы: уплотненное решение, нормальное решение с планированием по зависимостям, нормальное решение с планированием по уровням и субкритический вариант решения с планированием по уровням. Так как субкритическое решение было резервным вариантом для решения с планированием по уровням, команда проектирования исключила его из анализа риска.

Разуплотнение риска

Решение с планированием по уровням имело повышенный риск и критические импульсы на диаграмме; для устранения этой проблемы команда проектирования применила разуплотнение риска. Так как подходящая величина разуплотнения была неизвестна, команда проектирования попыталась разуплотнить его на 1 неделю, 2 недели, 4 недели, 6 недель и 8 недель и понаблюдать за поведением риска. В табл. 13.9 приведены значения риска для трех вариантов планировании и пяти точек разуплотнения.

Таблица 13.9. Значения риска для вариантов и точек разуплотнения

Вариант	Продолжительность (месяцы)	Риск возникновения критичности	Риск активности
Уплотненный	7,1	0,75	0,73
Планирование по зависимостям	7,8	0,70	0,70
Планирование по уровням	8,1	0,76	0,75
D1	8,3	0,60	0,65
D2	8,5	0,48	0,57
D3	9,0	0,42	0,46
D4	9,4	0,27	0,39
D5	9,9	0,27	0,34

На рис. 13.10 эти варианты и точки разуплотнения нанесены на временную шкалу. Риск возникновения критичности вел себя так, как ожидалось, и риск снизился с разуплотнением по некоторой логистической функции. Риск ак-

тивности также убывал с разуплотнением, но между двумя кривыми появился разрыв, потому что модель риска активности плохо реагировала на неравномерное распределение временных резервов. Вычисления, в результате которых были получены значения в табл. 13.9, решили эту проблему корректировкой выбросов временных резервов так, как описано в главе 11, то есть выбросы заменялись средним значением временных резервов плюс одно стандартное отклонение. В данном случае корректировки было попросту недостаточно. Корректировка временных резервов на половину стандартного отклонения обеспечила идеальное выравнивание кривых. Тем не менее команда проектирования решила ограничиться простым использованием кривой риска возникновения критичности, которая не требовала никаких корректировок. Команда пришла к выводу, что разуплотнение за пределами D4 было избыточным, потому что кривая риска выходила на горизонтальный уровень.

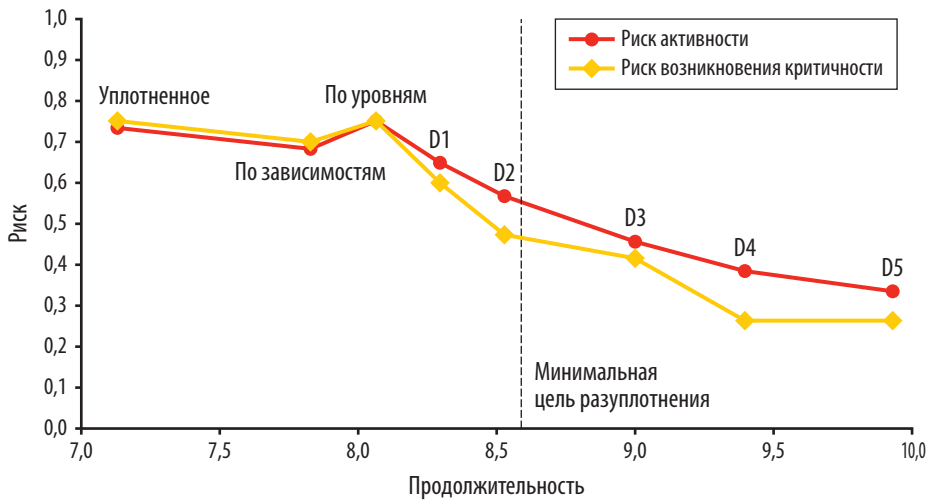


Рис. 13.10. Дискретные кривые риска

Со значениями в табл. 13.9 команда проектирования обнаружила модель полиномиальной корреляции для кривой риска с показателем R^2, равным 0,96:

$$\text{Риск} = 0{,}09t^3 - 2{,}28t^2 + 19{,}19t - 52{,}40,$$

где t измеряется в месяцах.

С моделью риска максимальный риск составил 7,4 месяца при значении риска 0,78. Эта точка располагалась между 7,8 месяца для решения с планированием по зависимостям и 7,1 месяца для уплотненного решения (рис. 13.11). Команда проектирования исключила уплотненное решение из рассмотрения, потому что оно выходило за пределы максимального риска. Даже решение с планированием

по зависимостям находилось на грани допустимого риска: при 7,8 месяца риск уже составлял 0,75 — максимальное рекомендуемое значение. У решения с планированием по уровням риск находился на комфортном уровне 0,68. Точка минимального риска приходилась на 9,7 месяца со значением риска 0,25.

В табл. 13.10 приведены значения риска для этих точек, а на рис. 13.11 эти точки наглядно представлены на кривой модели риска.

Таблица 13.10. Значения модели риска и точки, представляющие интерес

Вариант	Продолжительность (месяцы)	Модель риска
Уплотненный	7,1	0,75
Максимальный риск	7,4	0,78
Планирование по зависимостям	7,8	0,74
Планирование по уровням	8,1	0,68
Минимальные прямые затраты	8,46	0,56
D2	8,53	0,53
Минимальная цель разуплотнения	8,6	0,52
D3	9,0	0,38
Минимальный риск	9,7	0,25

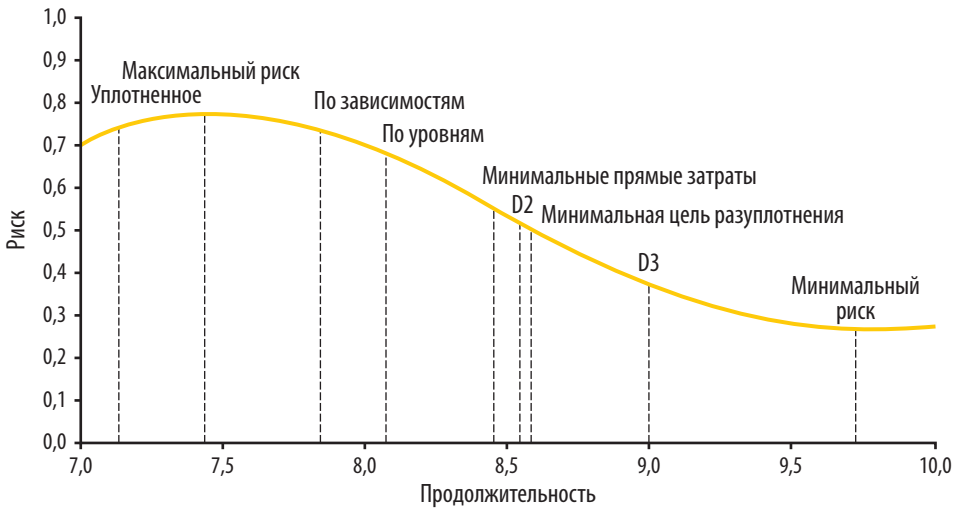


Рис. 13.11. Кривая модели риска и точки, представляющие интерес

Поиск цели разуплотнения

Используя методологию, рассмотренную в главе 12, команда проектирования вычислила минимальную цель разуплотнения риска (точку, в которой вторая производная кривой риска равна нулю): она находилась в точке 8,6 месяца со значением риска 0,52. Эта точка находилась между точками разуплотнения D2 и D3 (см. рис. 13.10), в результате чего точка справа от нее (D3) становится рекомендуемой целью разуплотнения. Риск при продолжительности D3 был равен 0,38 для модели риска — чуть менее реального значения 0,42 при D3. Хотя значение риска для цели разуплотнения может показаться низким (значительно меньшим идеального значения 0,5), оно соответствовало рекомендациям из главы 12 по разуплотнению проектов с планированием по уровням до 0,4 для компенсации присущего им риска.

Последним методом, примененным для поиска цели разуплотнения, было вычисление точки минимальных прямых затрат. Тем не менее прямые затраты точек разуплотнения были неизвестны.

Проанализировав рис. 13.8 и табл. 13.7, команда проектирования консервативно оценила, что три из четырех разработчиков должны продолжить работу при разуплотнении. Это позволило команде вычислить прямые затраты для расширения проекта до точки разуплотнения D5. Команда проектирования добавила эти дополнительные прямые затраты к известным прямым затратам решения с планированием по уровням; в результате была получена кривая прямых затрат и корреляционная модель с хорошей подгонкой:

$$\text{Прямые затраты} = 2{,}98t^2 - 50{,}42t + 244{,}53.$$

По формуле прямых затрат команда проектирования обнаружила точку минимальных прямых затрат 8,46 месяца непосредственно перед D2. Подстановка продолжительности 8,46 месяца в формулу риска дает риск 0,56. Разность продолжительности между минимальной точкой модели прямых затрат и нулевой точкой второй производной модели риска составила 1%, что подтвердило, что D3 является целью разуплотнения. Кстати говоря, минимальные прямые затраты были равны 31,4 человеко-месяца, хотя прямые затраты в D3 составили 32,2 человека (различия составляли всего 3%).

Повторное вычисление затрат

Рекомендованное решение D3 требовало, чтобы команда проектирования предоставила общие затраты для этой точки. Хотя прямые затраты были известны по предыдущей формуле, косвенные затраты были неизвестны на диапазоне разуплотнения. Команда проектирования смоделировала косвенные затраты для трех известных решений и получила простую прямую линию, которая описывалась следующей формулой:

$$\text{Косвенные затраты} = 7{,}27t - 30{,}01.$$

Команда проектирования сложила формулы прямых и косвенных затрат, чтобы получить формулу для общих затрат в системе:

$$\text{Общие затраты} = 2{,}98t^2 - 50{,}42t + 244{,}53 + 7{,}27t - 30{,}01 = \\ = 2{,}98t^2 - 43{,}5t + 214{,}52.$$

Согласно этой формуле, общие затраты для D3 составляют 67,6 человеко-месяца.

> **ПРИМЕЧАНИЕ** Команда проектирования использовала модели прямых затрат и риска для нахождения точек пересечения риска в проекте (хотя это и не делалось в реальном времени с заказчиком TradeMe). Это были точки 7,64 месяца при риске 0,77 (слишком рискованная точка пересечения) и 9,47 месяца при риске 0,27 (слишком безопасная точка пересечения). Эти точки хорошо соответствовали рекомендуемым значениям 0,75 и 0,3 соответственно, что лишний раз подтвердило правильность точек планирования, обсуждавшихся ранее.

Подготовка к анализу SDP

Лучшим вариантом планирования проекта до настоящего момента оказалась точка D3 — результат уплотнения решения с планированием по уровням на 1 месяц. Оно предоставило простой, достижимый проект при сокращенном риске и практически при минимальных прямых затратах. Относительно небольшие косвенные затраты сделали это решение оптимальным для проекта с точки зрения продолжительности, затрат и риска.

Кроме этой оптимальной точки, команда проектирования представила лицам, ответственным за принятие решений в компании, решение с планированием по зависимостям. Оно показывало, что любая попытка сокращения сроков кардинально повышала проектировочный риск и риск исполнения ввиду высокой сложности и нереалистичной ожидаемой эффективности команды.

Из-за потенциальной нехватки ресурсов команда проектирования пришла к выводу о необходимости включения субкритического решения, но только с адекватным разуплотнением. При повторении аналогичных действий для решения с планированием по уровням разуплотненное субкритическое решение обеспечило риск 0,47, продолжительность 11,8 месяца и общие затраты 79,5 человеко-месяца. Разуплотненное субкритическое решение было представлено для демонстрации как последствий от возможного недокомплектования проекта, так и того, что проект остается реализуемым, если возникнет такая необходимость.

Из-за повышенного риска не было смысла рассматривать неразуплотненные варианты решения с планированием по уровням и субкритического решения.

В табл. 13.11 приведена сводка вариантов планирования проекта, представленных командой проектирования на анализе SDP.

Таблица 13.11. Возможные варианты планирования проекта

Вариант	Продолжительность (месяцы)	Общие затраты (человеко-месяцы)	Риск	Сложность
Управляемый активностями	8	61	0,74	Высокая
Управляемый архитектурой	9	68	0,38	Низкая
С недокомплектованием	12	80	0,47	Низкая

Для целей представления команда проектирования переименовала варианты планирования, чтобы в них не было терминов типа «нормальное», «разуплотненное», «субкритическое» и «по уровням». В табл. 13.11 «Управляемый активностями» означает планирование по зависимостям, «управляемый архитектурой» — планирование по уровням, а «С недокомплектованием» — субкритическое решение.

Сложность в таблице обозначается простыми терминами вроде «Высокая» или «Низкая», а все числа, кроме значений риска, округлены. Таблица деликатно направляет ответственных за принятие решений к разуплотненному варианту с планированием по уровням.

14 Завершение

В предыдущих главах наше внимание было сосредоточено на технических аспектах планирования проекта. Конечно, планирование проекта можно рассматривать как техническую задачу проектирования. После нескольких десятилетий практического планирования проектов я понял, что это скорее образ мышления, а не обычный опыт. Ваша задача не ограничивается простым вычислением риска или затрат и попытками выполнить ваши обязательства. Вы должны стремиться к идеальному состоянию во всех аспектах проекта. Подготовьте меры противодействия всему, что проект может поставить перед вами, — для этого вам придется выйти за пределы механик и чисел. Вы должны взять на вооружение комплексный подход, который отражает вашу личность и отношение, особенности вашего общения с руководством и разработчиками, а также понимание последствий планирования для процесса разработки и жизненного цикла продукта. Идеи, которые я изложил для проектирования систем и планирования проектов в обеих частях книги, открывают путь на новый уровень мастерства в области разработки программных продуктов. Вам остается держать этот путь открытым, продолжать совершенствоваться, развивать эти идеи, культивировать собственный стиль и адаптироваться к обстоятельствам. Завершающая глава книги показывает, как можно подходить к этим аспектам, но что еще важнее — показывает, как двигаться дальше.

Когда планировать проект

На вопрос о том, когда следует заниматься планированием проекта, есть несколько возможных ответов. Самый прямолинейный ответ — «всегда». По сравнению с кошмарным состоянием дел во многих программных проектах преимущества, которые предоставляет планирование проекта, никаких сомнений не вызывают.

Меня как инженера всегда настораживают безапелляционные ответы типа «никогда» или «всегда». На вопрос о том, когда следует заниматься плани-

рованием проекта, стоит отвечать с точки зрения окупаемости. Сравните время и затраты на планирование проекта с преимуществами от построения системы самым быстрым способом, наименее затратным способом и самым безопасным способом. Так как планирование проекта занимает от нескольких дней до недели, с точки зрения окупаемости планирование большинства проектов легко оправдывается. Более того, чем больше масштаб проекта, тем больше усилий стоит направить на создание плана проекта, который предоставляет оптимальное решение. У больших и высокозатратных проектов даже небольшое отклонение от оптимальной точки может иметь огромные последствия в абсолютных терминах и почти наверняка превзойдет затраты на планирование проекта.

Другой ответ на вопрос о том, когда следует планировать проект, — «в любой ситуации, когда вы сталкиваетесь с жесткими сроками». Даже без уплотнения простое распределение эффективной команды по критическому пути нормального решения превзойдет любой другой подход, особенно по сравнению с проектами, которые пытаются строить систему итеративным методом.

Настоящий ответ

Последний ответ на вопрос о том, когда следует планировать проект, составляет самый важный раздел этой книги. Представьте, что у вас есть идея нового сверхпопулярного приложения, которое ждет огромный успех. Для его построения вам понадобится капитал для покрытия исходных затрат, от найма людей до оплаты времени облачных вычислений. Можно привлечь венчурный капитал в обмен на значительную долю в бизнесе, а затем несколько лет работать по 60 часов в неделю над проектом, который может завершиться неудачей. Также можно самостоятельно финансировать проект: продать дом, потратить пенсионный вклад и накопления и занять денег у друзей и семьи.

Если будет выбран вариант с самофинансированием, станете ли вы тратиться на планирование проекта? Будут ли затраты времени и усилий малыми или значительными? А может, вы решите, что у вас нет времени на планирование проекта? Скажете, что лучше просто взяться за дело и разобраться в процессе, или сделаете все необходимое, чтобы проверить проект на жизнеспособность, прежде чем остаться без денег и перспектив? Пропустите ли вы какие-либо методы или анализ планирования проекта? Даже если проект реализуем, захотите ли вы сознательно отказаться от планирования и выявления зон риска? Станете ли вы повторять все вычисления для пущей верности? Проведете ли вы изначальное планирование проекта, чтобы узнать, стоит ли вам продавать дом и увольняться с работы? В конце концов, если для проекта нужно 3 миллиона, а вы можете собрать только 2 миллиона, лучше выбрать дом вместо фирмы. То же самое можно сказать о продолжительности проекта. Если размер окна маркетинга составляет 1 год, а проект займет 2 года, то сделать ничего не удастся. А если выбран вариант с самофинансированием, то предпочли бы вы, чтобы

разработчики трудились по подробным инструкциям и не тратили скудные ресурсы, пытаясь самостоятельно разобраться в происходящем?

Теперь представьте проект, в котором руководитель несет личную ответственность за любые неудачи по соблюдению обязательств. Вместо того чтобы получить неплохую премию за выполнение проекта, в случае неудачи руководителю приходится из своего кармана оплачивать превышение затрат, а то и упущенные продажи, а также все договорные обязательства. Станет ли в такой ситуации руководитель сопротивляться планированию проекта или же, наоборот, будет настаивать на нем? Отклонит ли руководитель работу по планированию, потому что «у нас тут так не принято»? Потратит ли руководитель малые или значительные ресурсы на проектирование системы и планирование проекта, чтобы обеспечить соответствие обязательств с возможностями команды? Решит ли руководитель, что местонахождение мертвой зоны его не интересует? Откажется ли он от основательной проработки архитектуры, которая гарантирует, что план проекта не будет изменяться по ходу работы? Скажет ли руководитель, что раз так никто не работает, это достаточная причина для того, чтобы отказаться от планирования проекта?

Контраст очевиден. Когда компания платит, многие люди начинают относиться к происходящему безразлично и расслабленно. Они не желают мыслить самостоятельно, потому что намного проще догматически следовать общим практикам отрасли, которая переживает не лучшие времена, и использовать это как оправдание для растраты чужих денег. Чаще всего люди отговариваются тем, что у них нет времени, что процесс планирования проекта ошибочен или что планирование проекта — инженерное излишество. Тем не менее когда на кону оказываются их собственные деньги, те же люди становятся яростными сторонниками планирования проектов. Такие различия в поведении — прямое следствие отсутствия целостности: как личной, так и профессиональной. Поэтому правильный ответ на вопрос о том, когда следует планировать проект, звучит так: тогда, когда вы обладаете необходимой принципиальностью.

Как добиться успеха в жизни

Лучший совет по продвижению карьеры, который я могу вам дать, звучит так:

Относитесь к деньгам вашей компании как к своим собственным.

Все остальное не так важно. Многие руководители не могут отличить хорошую архитектуру от плохой, поэтому никогда не станут повышать или премировать вас на основании одной архитектуры. Но если вы относитесь к деньгам компании как к своим, если будете тщательно планировать проект, чтобы найти самый экономичный и безопасный способ построения системы, и если отклоняете любой другой образ действий — начальство это заметит. Демонстрируя

ФИНАНСОВЫЙ АНАЛИЗ

В большинстве проектов серьезного размера кто-то в какой-то момент должен решить, как оплачивать проект. Менеджеру проекта иногда даже приходится представлять данные ожидаемого темпа работ или оборота денежных средств. Это особенно важно для больших проектов. В таких проектах заказчик обычно не может выплатить круглую сумму в начале или в конце проекта, поэтому организация-разработчик должна финансировать разработку по некоторому графику платежей. В большинстве случаев при отсутствии информации о ходе проекта или структуре сети финансовое планирование превращается в смесь из домыслов, самообмана и функциональной декомпозиции платежей (то есть определенная сумма за каждую функцию). Часто это становится залогом провала. На самом деле финансовая сторона проекта не требует никаких домыслов. При минимальном объеме дополнительной работы план проекта может быть расширен в финансовый анализ проекта.

По распределению комплектования можно вычислить затраты для каждого временного сегмента проекта. Эти затраты представляются в виде накапливаемой суммы — в абсолютных или в относительных значениях (процентах). Вы даже можете представить сравнительные графики зависимости прямых и общих затрат от времени в числовом или графическом виде (для финансового планирования следует использовать денежные единицы вместо единиц трудозатрат, поэтому вы должны определить стоимость человеко-месяца для вашей организации).

Причина, по которой я упоминаю аспект финансового планирования проекта в книге, посвященной проектированию программных систем, не имеет особого отношения к финансам, какими бы ценными они ни были. В большинстве программных проектов люди, пытающиеся спроектировать систему и спланировать проект, применять передовые методы и соблюдать свои обязательства, сталкиваются с упорным сопротивлением, потому что остальные решительно настроены делать все худшим из возможных способов.

Где-то в организации имеется специалист по финансовому планированию, вице-президент или администратор, который должен принимать финансовые решения. Такие люди обладают значительными полномочиями и авторитетом, но часто действуют вслепую. Если вы объясните этим людям, что можете спроектировать финансовые подробности проекта до степени, описанной в тексте, они будут настаивать на этом. Конечно, способность построения графиков потоков денежных средств и затрат зависит от жизнеспособного плана проекта, который, в свою очередь, обусловлен наличием правильной архитектуры. Внезапно старший руководитель становится вашим главным союзником для правильной организации работы над проектом.

крайнее уважение к деньгам компании, вы заслужите ее уважение, потому что уважение всегда взаимно. И наоборот, люди не уважают тех, кто не проявляет уважения к ним. Когда вы отвечаете за свои действия и решения, ваша ценность в глазах руководства значительно возрастает. Раз за разом выполняя свои обязательства, вы заслужите доверие руководства. И когда представится следующая возможность, она будет доверена тому, кто доказал свое уважение к деньгам и времени компании, — вам.

Этот совет основан на моей собственной карьере. До того как мне исполнилось 30 лет, я возглавлял группу программной архитектуры в компании из списка «Fortune 100» в Кремниевой долине — месте самой яростной конкуренции в программной отрасли. Мой карьерный рост не имел особого отношения к моей квалификации как архитектора (как я уже говорил, это вообще не имеет особого значения). Тем не менее я всегда дополнял проектирование системы планированием проекта, и это имело решающее значение. Я относился к деньгам компании как к своим собственным.

Общие рекомендации

Не пытайтесь спроектировать часы.

После многих лет разочарований и крушений надежд в программных проектах люди, впервые сталкивающиеся с идеями планирования проектов, пленяются его точностью и приходят в восторг от его инженерных принципов. У них возникает искушение вычислять все до последнего знака, досконально уточнять каждое предположение и оценку, тем самым упуская суть основательного планирования проекта. Самое важное, что позволяет сделать планирование проекта, принятие обоснованных решений относительно проекта: стоит ли продолжать вообще, и если стоит, то по какому варианту. Выбранный вами план проекта всегда отличается от реальности, а фактическое исполнение проекта будет похоже на результат вашего планирования, но не будет полностью совпадать с ним. Менеджер проекта должен следить за ходом работы, часто сверяясь с планом и принимая меры по исправлению положения (см. приложение А).

Даже лучшее решение по планированию проекта всего лишь дает шансы на успех при исполнении — ничего более. Обратите внимание: «лучшее» в этом контексте означает решение, лучше всего соответствующее тому, что может произвести ваша команда (в категориях времени, затрат и риска), а не обязательно оптимально спроектированное.

Результат планирования проекта лучше представлять себе как солнечные, а не механические часы. Солнечные часы — исключительно простое устройство (вертикальная палка, воткнутая в землю), но их возможностей хватает для того, чтобы определить время с точностью до минуты (если известна

дата и широта). Механические часы могут определить время с точностью до секунды, но это гораздо более сложное устройство, которое работает только при идеальной настройке всех внутренних деталей. Результат вашей работы по планированию проекта должен быть точным до такой степени, чтобы вы могли примерно определить, какие обязательства можете принять. Оптимальные точные решения, выверенные до мельчайших подробностей, — дело хорошее, но нормальные практичные решения — необходимость.

Архитектура и оценки

Никогда не планируйте проект без основательной архитектуры, инкапсулирующей все нестабильности.

Без правильно проработанной архитектуры структура системы в какой-то момент изменится. Эти изменения означают, что вы будете строить другую систему, а спроектированное решение станет недействительным. И когда это произойдет, будет совершенно неважно, что изначально у вас был идеально спланированный проект. Как объяснялось в части I книги, необходимо выделить время на то, чтобы разобраться с нестабильностями независимо от того, будете вы применять структуру Метода или нет.

В отличие от архитектуры, оценки и конкретные ресурсы вторичны для хорошего планирования проекта. Продолжительность проекта определяется топологией сети (которая обусловлена архитектурой), а не способностями разработчиков или (до некоторой степени) колебаниями отдельных оценок. Оценки, значительно отличающиеся от реальности, могут кардинально повлиять на проект. Тем не менее при условии, что оценка более или менее верна, некоторое отклонение реальной продолжительности в ту или иную сторону уже не столь важно. В проекте сколько-нибудь приличного размера содержатся десятки активностей, оценки которых могут быть смещены в любую сторону. В общем и целом эти смещения обычно компенсируют друг друга. То же относится к способностям разработчиков. Производительности худшего и лучшего разработчика в мире очень сильно различаются, но если у вас работают нормальные разработчики, ситуация выравнивается. Проявить творческий подход к проектированию, вовремя распознать ограничения и найти обходные пути вокруг потенциальных проблем важнее, чем добиться идеальной точности каждой оценки.

Отношение к планированию

Идеи, представленные в книге, не следует применять догматически.

Средства планирования проекта следует адаптировать к конкретным обстоятельствам без ущерба для конечного результата. В книге я постарался продемонстрировать вам некоторые возможности, чтобы разжечь ваше природ-

ное любопытство, подтолкнуть вас действовать творчески и вести за собой других.

По возможности не пытайтесь планировать проект тайно от других. Артефакты и видимый процесс планирования формируют доверие у ответственных за принятие решений. Если ключевые участники спрашивают, объясните им, что вы делаете и почему поступаете именно так.

Альтернативность

Общайтесь с руководством на языке Альтернативности.

Во всех взаимодействиях с руководством используйте стиль, который я называю Альтернативностью: краткое описание вариантов, из которых руководство может выбрать нужный, с приведением объективной оценки каждого из вариантов. Такой подход хорошо соответствует основной концепции планирования проекта: не существует «единственно правильного» проекта. Всегда есть несколько вариантов построения и реализации любой системы. Каждый вариант представляет некоторую комбинацию времени, затрат и риска. Таким образом, вы должны создать несколько вариантов, из которых руководство выберет наиболее подходящий.

Суть хорошего управления — выбор правильного варианта. Более того, наличие нескольких вариантов расширяет возможности. В конце концов, если нет вариантов, то и руководитель не нужен. Руководители, у которых нет вариантов для выбора, вынуждены оправдывать свое присутствие принятием произвольных решений.

Без поддерживающего плана проекта такие неестественные варианты всегда приводят к плохим результатам. Чтобы избежать опасности, вы должны представить руководству набор жизнеспособных вариантов планирования проекта, заранее отобранных вами. Например, в главе 11 рассматривались до 15 вариантов планирования проекта, но соответствующий анализ SDP включал всего 4 варианта. Впрочем, не стоит злоупотреблять Альтернативностью. Чрезмерное обилие вариантов беспокоит людей — такая ситуация называется *парадоксом выбора*[1]. Этот парадокс происходит от страха упустить лучший вариант, который вы не выбрали, даже если выбранный вариант был достаточно хорош.

Несколько рекомендаций по выбору количества вариантов:

- Двух вариантов мало — слишком близко к полному отсутствию вариантов.

[1] Barry Schwartz, The Paradox of Choice: Why More Is Less (Ecco, 2004). (На русском: *Шварц Барри*. Парадокс выбора. Как мы выбираем и почему «больше» значит «меньше». М.: Добрая книга, 2005. — *Примеч. ред.*)

- Три варианта — идеально: большинство людей может легко выбрать из трех вариантов.
- Четыре варианта — нормально при условии, что по крайней мере один из них (а возможно, и два) является очевидной ошибкой.
- Пять вариантов — слишком много, даже если все варианты хороши.

Уплотнение

Не превышайте 30% уплотнения.

Какой бы способ уплотнения проекта вы ни выбрали, 30-процентное сокращение срока — максимальное уплотнение, на которое можно рассчитывать, если вы начинаете с хорошо проработанного нормального решения. Проекты с чрезмерным уплотнением обычно страдают от высокого риска исполнения и планирования. Если вы только начинаете осваивать средства планирования проектов и строите доверительные отношения с командой, избегайте решений с уплотнением более 25%.

Понимание проекта

Всегда проводите уплотнение проекта, даже если вероятность выбора какого-либо из уплотненных решений низка.

Уплотнение раскрывает истинную природу и поведение проекта и всегда дает информацию для лучшего понимания проекта. Уплотнение позволяет смоделировать кривую «время-затраты» для проекта, а формулы для затрат и риска пригодятся, когда вам понадобится оценить эффект изменений графика. Возможность быстро и убедительно определить вероятные последствия запросов на внесение изменений исключительно ценна. Если ее нет, остается полагаться на чутье, что неизменно приводит к конфликтам.

Даже если вы подозреваете, что поступивший запрос неразумен, ответ «нет» — особенно руководителю, наделенному властью, — не способствует вашей карьере. Единственный способ сказать им «нет» — заставить их самих сказать «нет». Представляя объективные последствия для сроков, затрат и риска, вы немедленно поднимаете на поверхность то, что до этого знали на интуитивном уровне; это позволяет вести объективное обсуждение, свободное от эмоций. Без чисел и метрик возможны любые аргументы. Незнание реальности — не грех, в отличие от профессиональной некомпетентности. Если ответственным за принятие решений известны числовые данные, которые противоречат их обязательствам перед заказчиками, но они все еще настаивают на этих обязательствах, они фактически сознательно идут на обман. Поскольку такая ответственность неприемлема, при наличии объективных чисел они найдут способы аннулировать свои обязательства или изменить ранее «неизменяемые» даты.

Уплотнение с лучшими ресурсами

Проводите уплотнение за счет использования лучших ресурсов осторожно и осмотрительно.

Если вы задействуете лучшие ресурсы, правильное планирование проекта крайне важно для того, чтобы знать, где применять их. Как бы привлекательно ни выглядело уплотнение за счет использования лучших ресурсов, оно может возыметь обратный эффект. Прежде всего, ценные специалисты встречаются редко, так что лучшие ресурсы, которые могут понадобиться для выполнения ваших обязательств, могут оказаться недоступными. Ожидание создает задержки и противоречит исходной цели уплотнения. Даже когда лучшие ресурсы станут доступными, они могут ухудшить ситуацию, потому что их применение для уплотнения критического пути может привести к появлению нового критического пути. Так как ресурсы распределяются на основании временных резервов и свободных мощностей, появляется риск того, что на новом критическом пути будут работать *худшие* разработчики.

Даже при назначении на ранее критические активности лучшие ресурсы нередко простаивают, ожидая, пока их догонят другие активности и разработчики в проекте. Это приводит к снижению эффективности проекта. Чтобы избежать подобных ситуаций, вам понадобится большая команда, которая может уплотнять другие пути за счет параллельной работы. Увеличение размера команды сокращает эффективность и повышает затраты. Наконец, сжатие с использованием лучших ресурсов часто требует двух и более таких героев для уплотнения нескольких критических или околокритических путей, чтобы преимущества уплотнения проявились на практике.

При назначении лучших ресурсов не следует действовать вслепую (например, назначая лучший ресурс на все текущие критические активности). Оцените, на каком сетевом пути ресурсы принесут наибольший выигрыш, определите эффект на других путях и даже опробуйте различные комбинации между цепочками. Возможно, вам придется несколько раз переназначить лучший ресурс на основании изменений на критическом пути. Также необходимо учитывать как размеры активностей, так и их критичность. Например, у вас может быть большая некритическая активность с высоким уровнем неопределенности, которая может легко загубить проект. Назначение лучших ресурсов сократит риск и в конечном итоге поможет выполнить обязательства.

Усечение нечеткой начальной стадии

Простейший способ уплотнения проекта — усечение исходных активностей проекта в нечеткой начальной стадии.

Хотя никакой проект не может быть ускорен свыше своего критического пути, к начальной стадии это правило не относится. Ищите возможности организо-

вать параллельную работу в начальной стадии над подготовительными или оценочными задачами. Это приведет к уплотнению начальной стадии (а следовательно, и проекта) без каких-либо изменений в оставшейся части проекта. Например, на рис. 14.1 показан проект (верхняя диаграмма) с длинной начальной стадией. Начальная стадия содержит ряд важных технологических и проектировочных решений, которые должны быть приняты архитектором до того, как выполнение проекта сможет продолжиться. Привлечение второго архитектора в качестве подрядчика для этих двух решений позволило сократить продолжительность начальной стадии на треть (нижняя диаграмма на рис. 14.1).

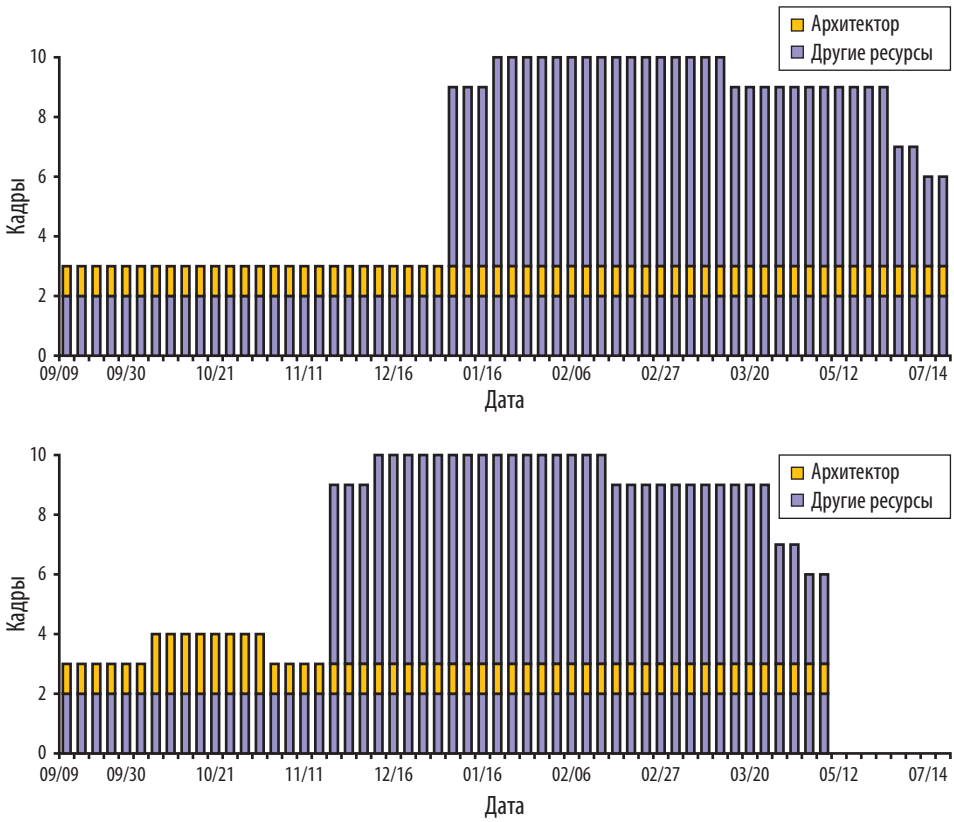


Рис. 14.1. Усечение начальной стадии с привлечением второго архитектора

Планирование и риск

Действуйте на упреждение с использованием временного резерва.

Индекс риска показывает, пойдет ли проект насмарку при столкновении с первым препятствием или же сможет воспользоваться этим препятствием, чтобы

получить дополнительную информацию и лучше приспособить план к реальности. Наличие достаточного временного резерва (на что указывает более низкий риск) даст вам возможность добиться успеха перед лицом непредвиденных обстоятельств.

Я также обнаружил, что потребность проекта во временном резерве имеет не только физическую, но и психологическую основу. Физическая необходимость очевидна: временной резерв потребляется для того, чтобы вы могли справиться с изменениями и переместить ресурсы между активностями. Психологическая необходимость связана с душевным спокойствием всех участников. В проектах с достаточным временным резервом люди избавлены от лишнего стресса; они концентрируются на своих задачах и справляются с ними.

Поведение, а не конкретные значения

В главе 10 было предложено значение 0,5 как минимальная цель разуплотнения и значение 0,3 как минимальный уровень риска. Какими бы полезными ни были эти рекомендации, при анализе кривой риска проекта следует помнить, что поведение важнее конкретных значений. При разуплотнении проекта ищите точку перегиба риска, а не значение 0,5. Какие-то факторы могут сместить всю кривую риска выше или ниже, но точка перегиба риска все равно должна присутствовать. Это особенно справедливо в тех случаях, когда нормальное решение уже обладает высоким риском. Возможно, вам придется уплотнить проект, но для этого будет использоваться поведение точки перегиба.

Реализация планирования проекта

Планирование проекта — активность, уделяющая особое внимание подробностям. К акту планирования проекта следует относиться как к очередной нетривиальной деятельности, которую необходимо спланировать и тщательно подготовить. Иначе говоря, вы должны распланировать сам процесс планирования проекта и даже применить для этого средства планирования проектов.

Чтобы вам было проще взяться за работу, мы приведем список стандартных действий по планированию:

1. Сбор базовых сценариев использования.
2. Проектирование системы, построение цепочек вызовов и списка компонентов.
3. Построение списка активностей, не связанных с программированием.
4. Оценка продолжительности и необходимых ресурсов для всех активностей.
5. Общая оценка проекта с использованием широкополосного метода и/или специализированных средств.

6. Проектирование нормального решения.
7. Анализ решения с ограниченными ресурсами.
8. Поиск субкритического решения.
9. Уплотнение с использованием лучших ресурсов.
10. Уплотнение с использованием параллельной работы.
11. Уплотнение с использованием изменения активностей.
12. Уплотнение до минимальной продолжительности.
13. Выполнение анализа результативности, эффективности и сложности.
14. Построение кривой «время-затраты».
15. Разуплотнение нормального решения.
16. Повторное построение кривой «время-затраты».
17. Сравнение кривой «время-затраты» с общей оценкой проекта.
18. Числовое выражение и моделирование риска.
19. Нахождение зон включения/исключения и зон риска.
20. Выявление реализуемых вариантов.
21. Подготовка к анализу SDP.

Хотя некоторые из этих активностей могут выполняться параллельно с другими активностями, между активностями проектирования системы и планирования проекта существуют взаимные зависимости. Следующим логическим шагом должно стать планирование самого процесса планирования проекта с использованием простой диаграммы сети и даже вычислением общей продолжительности работы. На рис. 14.2 изображена диаграмма сети для процесса планирования. Вероятный критический путь можно определить по типичным продолжительностям активностей. Если планированием проекта занимается один архитектор, то диаграмма будет представлять собой длинную цепочку. Если же архитектору кто-то помогает или если архитектор ожидает некоторой информации, диаграмма подскажет, какие активности могут выполняться параллельно.

Активности 6, 7, 8, 9, 10, 11 и 12 в списке (выделены синим цветом на рис. 14.2) являются конкретными решениями планирования проекта. Каждую из них можно дополнительно разбить на следующие задачи:

1. Определение предпосылок планирования.
2. Сбор требований к комплектованию.
3. Анализ и пересмотр списка активностей, оценок и ресурсов.
4. Выбор зависимостей.
5. Изменение сети с учетом ограничений.

6. Изменение сети для сокращения сложности.
7. Назначение ресурсов для активностей и переработка сети.
8. Построение диаграммы сети.
9. Вычисление пологой S-образной кривой.
10. Вычисление диаграммы распределения планирования.
11. Изменение предпосылок планирования и переработка сети.
12. Вычисление элементов затрат.
13. Анализ временных резервов.
14. Вычисление риска.

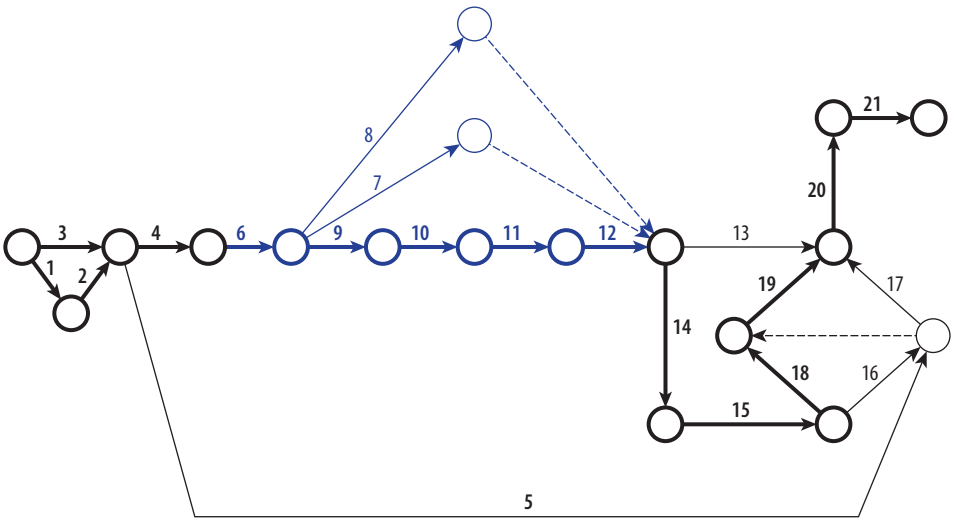


Рис. 14.2. Планирование процесса планирования проекта

Перспективы

В любой системе важно различать объем работы и масштаб. Архитектура в программной системе должна быть всеобъемлющей как по масштабу, так и по времени. Она должна включать все необходимые компоненты и быть правильной как в текущий момент, так и в далеком будущем (при условии, что природа бизнеса останется неизменной). Вы должны избегать очень дорогостоящих и дестабилизирующих изменений, которые являются результатом несовершенного проектирования. В том, что касается объема работы, архитектура должна быть крайне ограниченной. В части I книги объяснялось, как выработать основательную декомпозицию на основе нестабильности за период от

нескольких дней до недели, даже в больших системах. Для этого необходимо уметь правильно решать такие задачи, но, конечно, для этого также потребуется опыт и практика.

По сравнению с архитектурой проектирование, особенно подробное проектирование сервисов или пользовательского интерфейса *Клиентов*, занимает больше времени и имеет ограниченный масштаб. Проработка проектирования всего нескольких взаимодействующих сервисов может занять несколько недель.

Наконец, программирование является самой трудоемкой частью проекта, которая к тому же наиболее ограничена по масштабу. Разработчики никогда не должны программировать более одного сервиса за раз, и они тратят значительное время на тестирование и интеграцию всех сервисов.

На рис. 14.3 масштаб сравнивается с объемом работы для программного проекта в качественном виде. Вы видите, что масштаб и объем работ буквально являются обратными характеристиками. Если какая-то часть проекта имеет более широкий масштаб, то она сужается по объему работ, и наоборот.

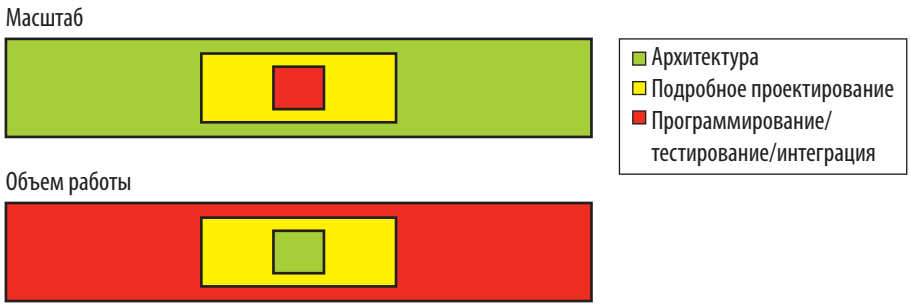


Рис. 14.3. Масштаб и объем работы в программной системе

Подсистемы и график

В главе 3 обсуждается концепция отображения подсистем на вертикальные сегменты архитектуры. В большом проекте таких подсистем может быть несколько. Эти подсистемы должны быть относительно хорошо изолированы и независимы друг от друга. Каждая подсистема имеет собственный набор активностей (например, подробное проектирование и построение). В последовательном проекте подсистемы следуют друг за другом, как показано на рис. 14.4.

Следует учесть, что подсистемы всегда проектируются и строятся в контексте существующей архитектуры. Распределение объема работы на рис. 14.4 все еще соответствует рис. 14.3.

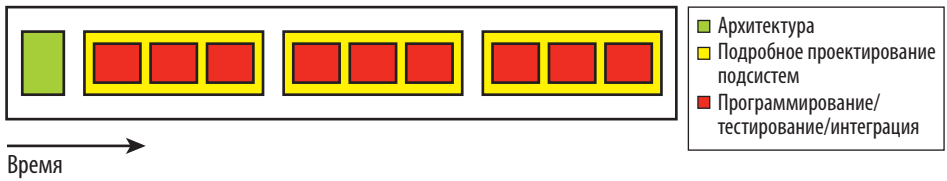


Рис. 14.4. Жизненный цикл последовательного проекта

Возможно, вам удастся уплотнить проект и начать работать параллельно. На рис. 14.5 изображены два режима параллельной разработки подсистем, совмещенные на временной оси.

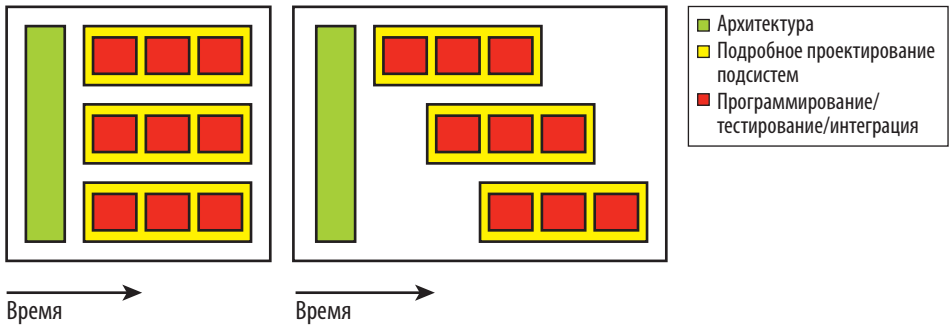


Рис. 14.5. Жизненные циклы параллельного проекта

Выбор параллельного жизненного цикла зависит от уровня зависимостей между подсистемами архитектуры. На рис. 14.5 жизненный цикл в правой части выстраивает подсистемы так, чтобы они перекрывались на временной оси. В этом случае вы начинаете строить подсистему после завершения реализации интерфейсов, от которых она зависит. После этого вы можете работать над оставшейся частью подсистемы параллельно с предыдущей. Возможно даже создание полностью параллельных конвейеров вроде изображенного в левой части на рис. 14.5. В этом случае каждая подсистема строится независимо от других подсистем и параллельно с ними с минимальной интеграцией.

Из рук в руки

Состав и структура команды оказывают значительное влияние на планирование проекта. Здесь под структурой команды понимается соотношение численности старших и младших разработчиков. Многие организации (и даже отдельные люди) определяют старшинство по опыту практической работы (в годах). Я использую другое определение: старшие разработчики — те, которые способ-

ны выполнять подробное проектирование сервисов; младшие разработчики на это не способны. Подробное проектирование происходит после основной архитектурной декомпозиции системы на сервисы. Для каждого сервиса подробное проектировочное решение включает проектирование открытых интерфейсов (или контрактов) сервисов, их сообщений и контрактов данных, а также такие внутренние подробности, как иерархии классов или средства безопасности.

Обратите внимание: старшие разработчики не определяются как разработчики, которые умеют или знают, как выполнять подробное проектирование. Вместо этого старшие разработчики *способны* выполнить подробное проектирование после того, как вы им покажете, как это правильно делать.

Джуниоры

Если у вас в команде только младшие разработчики, архитектор должен предоставить им подробно спроектированные сервисы. Такой подход непропорционально увеличивает рабочую нагрузку архитектора. Например, в 12-месячном проекте от 3 до 4 месяцев может быть потрачено просто на подробное проектирование.

Работа по подробному проектированию может выполняться архитектором в начальной стадии или в то время, когда разработчики занимаются построением некоторых сервисов. Оба варианта плохи.

Проработка всех подробностей сервисов на ранней стадии — непростое дело, а умение заранее предвидеть все тонкости взаимодействий всех сервисов требует очень высокой квалификации. Спроектировать заранее можно несколько сервисов, но не все. Настоящая проблема заключается в том, что подробное проектирование на начальной стадии просто занимает слишком много времени. Руководство вряд ли осознает важность подробного проектирования и будет недовольно кривиться от перспективы расширять начальную стадию для выполнения этой работы. Скорее всего, это кончится тем, что руководство заставит вас передать проработку архитектуры джуниорам и обречет проект на неудачу.

Проектирование сервисов на ходу параллельно с построением разработчиками сервисов, которые уже были спроектированы архитектором, может сработать. Однако если архитектор будет перегружен подробным проектированием, он станет узким местом системы, что может привести к существенному замедлению проекта.

Сеньоры

Старшие разработчики играют исключительно важную роль в решении проблемы подробного проектирования. Если старшие разработчики еще не умеют

выполнять работу по подробному проектированию, после непродолжительного обучения они справятся с ней.

При взаимодействии со старшими разработчиками архитектор может передать им проектировочное решение вскоре после анализа SDP, предоставив только схематичное описание сервисов в форме описаний интерфейсов или просто предложив определенный паттерн проектирования. Подробное проектирование теперь выполняется как часть работы над каждым отдельным сервисом, а архитектору останется только проанализировать результат и доработать его по мере надобности. Собственно, единственная причина для оплаты дополнительных сеньоров — возможность поручения им подробного проектирования. Это самый безопасный способ ускорения любого проекта, потому что он уплотняет график без изменений на критическом пути, которые увеличивают риск исполнения или создают узкие места. Короткие проекты требуют меньших затрат, так что в конечном итоге сеньоры обходятся дешевле джуниоров.

Сеньоры в роли архитекторов-джуниоров

Основная проблема с поручением подробного проектирования старшим разработчикам — их дефицит. У вас может быть один сеньор, два или три — но не вся команда. Если вы оказались в такой ситуации, не используйте одного-двух сеньоров в качестве разработчиков. Вместо этого измените процесс, чтобы они занимались в основном работой по подробному проектированию. На рис. 14.6 показано, как выглядит этот процесс.

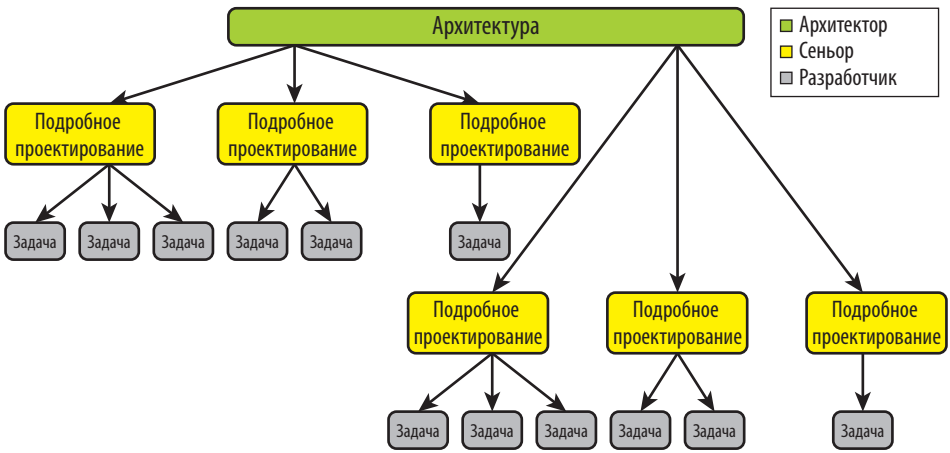


Рис. 14.6. Параллельная работа с архитекторами-джуниорами

Архитектор должен предоставить подробную архитектуру, как обсуждалось в части I этой книги. Архитектура не изменяется за время жизни системы,

а построение всегда осуществляется в контексте этой архитектуры. Построение архитектуры происходит в начальной стадии проекта. Начальная стадия также может содержать подробное проектирование первой группы сервисов. Это подробное проектирование осуществляется старшими разработчиками и используется для обучения и повышения их квалификации под руководством архитектора. В сущности, сеньоры при этом превращаются в архитекторов-джуниоров.

После того как подробное проектирование сервисов будет завершено, младшие разработчики могут подключиться к работе и заняться построением сервисов. Любое уточнение решения, каким бы тривиальным оно ни было, заставляет джуниоров советоваться со старшими разработчиками, которые проектировали сервис. После того как построение всех сервисов будет завершено, младшие разработчики проводят анализ кода с сеньорами (а не со своими младшими коллегами), после чего выполняется интеграция и тестирование с другими младшими разработчиками. В это время старшие разработчики занимаются подробным проектированием следующей группы сервисов. Каждое проектировочное решение анализируется архитектором перед тем, как оно будет передано джуниорам.

Такой способ организации работы — лучший и единственный способ снижения рисков, связанных со взаимодействием с младшими разработчиками. Очевидно, он также потребует тщательного планирования проекта. Вы должны точно знать, сколько сервисов вы сможете спроектировать заранее и как синхронизировать передачу работы с построением. Также необходимо добавить явные активности по подробному проектированию сервисов и даже дополнительные точки интеграции для решения проблемы риска, связанного с подробным проектированием сервисов.

На практике

Планирование проекта, как и проектирование системы, требует практики. От профессионалов — от адвокатов до врачей и пилотов — прежде всего ожидают досконального знания дела и умения идти до конца. В боевых условиях каждый специалист возвращается к своему уровню обученности. К сожалению, в отличие от проектирования систем, архитекторы обычно не знают о планировании проектов и не проходят подготовку в этой области, несмотря на то что планирование проекта исключительно важно для его успеха, и, как упоминалось в главе 7, отвечает за него архитектор.

Необходимость практики в области планирования проектов имеет два дополнительных аспекта. Во-первых, планирование проектов — огромная тема. В книге приведены основные знания, необходимые для современного архитектора (как в области проектирования систем, так и в области планирования

проектов). По количеству страниц планирование проекта в книге вдвое превосходит проектирование систем. Возможно, к этому моменту у вас уже появилось чувство, что вы заглядываете в бездонную пропасть. Невозможно усвоить и правильно применять концепции, представленные в книге, без практики и обучения. Попытки научиться планированию проектов на примере реальных проектов на рабочем месте не только ведут к неприятностям, но и противоречат здравому смыслу. Хотели бы вы стать первым пациентом врача — вчерашнего выпускника медицинского института? А лететь на самолете, которым управляет неопытный пилот? Гордитесь ли вы своей первой программой?

Во-вторых, результаты планирования проектов во многих случаях противоречат интуиции. Практика нужна не только для освоения огромного массива знаний, но и для формирования новой интуиции. К счастью, навыки планирования проектов можно приобрести, на что указывает быстрое и очевидное повышение качества планирования и показатели успеха тех, кто не пренебрегает практикой.

В главе 2 подчеркивается важность практики проектирования систем. Всегда сочетайте практику проектирования системы с практикой планирования проекта при его построении. Начните с простого нормального решения. Тренируйтесь до того момента, пока вы не начнете уверенно находить нормальные решения для своих учебных систем. После этого дорабатывайте найденный вариант и ищите лучшее решение с точки зрения графика, затрат и риска.

Проанализируйте свои прошлые проекты. Пользуясь имеющейся информацией, попытайтесь реконструировать фактический план проекта и сравните его с тем, что должно быть сделано. Идентифицируйте предположения планирования, классические ошибки и правильные решения. Подготовьтесь к анализу SDP: перечислите все решения, которые бы вы предложили, если бы это было возможно. Обратитесь к текущему проекту. Сможете ли вы составить список активностей, сформировать правильные оценки на основании того, чем сейчас занимается команда, вычислить истинный график и затраты? Каков текущий уровень риска? Что потребуется для разуплотнения проекта? Какой уровень уплотнения будет приемлемым?

Когда вы решите, что все сделано правильно, снова поднимите планку и найдите возможности для улучшения этих результатов. Никогда не почивайте на лаврах. Разработайте новые методы, сформируйте собственный стиль, станьте экспертом и настоящим сторонником планирования проектов.

Подведение итогов планирования проекта

Подведение итогов (дебрифинг) недостаточно широко применяется в программной отрасли, хотя это чрезвычайно эффективный метод с фантастической эффективностью. Краткое изложение вашей работы по планированию

проекта и ее результатов предоставляет возможность обменяться информацией, полученной для разных проектов и ролей, чтобы каждый участник мог учиться на опыте других. Все, что для этого потребуется, — навыки самопроверки, анализа и желание совершенствоваться. Проводите разбор по каждому своему проекту, подведение итогов должно стать частью жизненного цикла каждого проекта. Анализируйте как каждый проект в целом, так и каждую подсистему или контрольную точку. Чем более заметное место будет занимать подведение итогов в вашем рабочем процессе, тем вероятнее то, что вы сможете извлечь из него реальную пользу.

Набор тем, рассматриваемых при подведении итогов, зависит от того, что вы считаете важным и нуждающимся в улучшении. К их числу могут относиться следующие вопросы:

- **Оценки и точность.** Для каждой активности спросите себя: насколько точными оказались исходные оценки при сравнении с фактической продолжительностью и сколько раз вам приходилось корректировать оценки (и в каком направлении)? Наблюдается ли очевидная закономерность, которую можно было бы внедрить в будущие проекты для повышения качества оценок? Просмотрите исходный список активностей; посмотрите, что вы упустили из виду и что было лишним. Проведите вычисления и определите, до какой степени ошибки в оценках компенсировали друг друга.
- **Эффективность и точность планирования.** Сравните точность исходной оценки для проекта в целом с результатами подробного проектирования, а также фактической продолжительностью и затратами. Насколько точной была ваша оценка результативности команды? Было ли необходимым разуплотнение риска, и если было — не было ли оно избыточным (или недостаточным)? Наконец, был ли уплотненный проект жизнеспособным и как менеджер проекта и команда справлялись с его сложностью?
- **Индивидуальная и командная работа.** Насколько хорошо участники команды работают в коллективе или в одиночку? Не было ли среди них «слабых звеньев»? Можно ли повысить производительность команды в будущем за счет использования улучшенных инструментов или технологий? Своевременно ли команда сообщала о возникших проблемах? Насколько хорошо участники команды понимали план и свою роль в нем?
- **Чего следует избегать и что следует улучшить в следующий раз.** Составьте приоритетный список всех ошибок и проблем, с которыми сталкиваются люди, процессы, проектирование и технологии. Для каждого пункта укажите, как можно было бы быстрее обнаружить проблему или обойти ее. Перечислите действия, которые создали проблемы, и действия, которые следовало бы выполнить. Также в список следует включить опасные ситуации, которые в итоге не принесли прямого вреда.

- **Повторяющиеся проблемы из предыдущих разборов.** Один из лучших путей к совершенствованию — умение избегать старых ошибок и предотвращать возникновение известных проблем. Если одни и те же ошибки возникают в проекте за проектом, это плохо отразится на всех участниках. Скорее всего, многократное возникновение одной проблемы объясняется какой-то серьезной причиной. Тем не менее повторяющиеся ошибки необходимо устранить, несмотря на все трудности.
- **Ориентация на качество.** В какой степени ваш проект был ориентирован на качество? Насколько этот фактор был связан с успехом?

Важно проводить подведение итогов даже в успешных проектах, в которых обязательства были выполнены. Вы должны знать, почему проект завершился успешно: просто потому, что вам повезло, или благодаря качественному проектированию системы и планированию проекта. Даже если проект завершился успешно, можно ли было справиться лучше? И каким образом сохранить то, что было сделано правильно, в будущих проектах?

О качестве

В общем смысле эта книга посвящена качеству. Единственная цель наличия серьезной архитектуры — получение наименее сложной системы из всех возможных. Тем самым обеспечивается построение качественной системы, более простой в тестировании и сопровождении. Бесспорный факт: качество ведет к производительности, а обязательства по срокам и бюджету почти невозможно выполнить, если продукт полон дефектов. Если команда проводит меньше времени за выявлением проблем, то она проводит больше времени за созданием ценности. Хорошо спроектированные системы и проекты — единственный способ завершения работы в срок.

В любой программной системе качество зависит от наличия проектировочного решения, которое включает критические активности контроля качества как неотъемлемую часть проекта. Ваш план проекта должен учитывать активности контроля качества как по времени, так и по ресурсам. Не ищите легких путей, если целью планирования проекта является быстрое и чистое построение системы.

Побочный эффект планирования проекта заключается в том, что для хорошо спланированных проектов характерен низкий уровень стресса. Если проект располагает необходимым временем и ресурсами, люди уверены в своих способностях и в руководстве проекта. Они знают, что срок реален, а каждая активность учтена. Люди, находящиеся под меньшим давлением, уделяют больше внимания подробностям; ничто не остается незамеченным, а результат получается более качественным. Кроме того, хорошо спланированные проекты максимизируют

эффективность команды. Этот факт вносит свой вклад в качество: команда быстрее выявляет, изолирует и исправляет дефекты наименее затратным способом.

Ваша работа по проектированию системы и планированию проекта должна мотивировать команду к производству наиболее качественного кода. Вы увидите, что успех вызывает привыкание: после того, как люди привыкнут работать правильно, они начнут гордиться своей работой и уже не вернутся к прежнему. Никому не нравятся среды с высоким стрессом, для которых характерны низкое качество, трения и взаимные обвинения.

Активности контроля качества

Планирование проекта всегда должно учитывать элементы и активности контроля качества. К их числу относятся:

- *Тестирование уровня сервисов.* При оценке продолжительности и объема работы для каждого сервиса проследите за тем, чтобы оценка включала время, необходимое для написания плана тестирования сервиса, проведения модульных тестов по плану и выполнения интеграционного тестирования. При необходимости добавьте время, необходимое для проведения интеграционного тестирования, к регрессионному тестированию.
- *План тестирования системы.* Проект должен содержать явную активность, в которой квалифицированные инженеры по тестированию пишут план тестирования. Включаются все возможные способы нарушения работоспособности системы и проверка того, что они не работают.
- *Оснастка для системного тестирования.* Проект должен содержать явную активность, в которой квалифицированные инженеры по тестированию разрабатывают полнофункциональную тестовую оснастку.
- *Системное тестирование.* Проект должен содержать явную активность, в которой тестировщики, занимающиеся контролем качества, выполняют план тестирования с применением тестовой оснастки.
- *Ежедневные проверки общей работоспособности.* В составе косвенных затрат проекта вы должны ежедневно строить чистую сборку разрабатываемой системы, активировать ее и (в переносном смысле) пускать воду по трубам. Проверка работоспособности такого рода выявляет проблемы в связующем коде — например, дефекты создания экземпляров, сериализации, сетевого кода, тайм-аутов, безопасности и синхронизации. Сравнение результатов с результатами вчерашней проверки позволяет быстро обнаружить дефекты в связующем коде.
- *Косвенные затраты.* Качество не дается бесплатно, но обычно окупается, потому что дефекты обходятся ужасающе дорого. Вы должны правильно

учесть необходимые вложения в качество, особенно в форме косвенных затрат.

- *Тестирование сценариев автоматизации*. Автоматизация тестов должна стать явной активностью в проекте.
- *Проектирование и реализация регрессионного тестирования*. Проект должен включать подробное регрессионное тестирование, которое обнаруживает дестабилизирующие изменения в момент их возникновения в системе, подсистемах, сервисах и всех возможных взаимодействиях. Тем самым предотвращается каскадный эффект от новых дефектов, внесенных при исправлении существующих дефектов или простом внесении изменений. Хотя проведение регрессионного тестирования на постоянной основе часто рассматривается как косвенные затраты, проект должен содержать активности для написания регрессионного тестирования и его автоматизации.
- *Рецензирование на системном уровне*. В главе 9 рассматривается необходимость проведения обширного партнерского рецензирования на уровне сервисов. Так как дефекты могут возникнуть везде, рецензирование следует расширить на системный уровень. Основная команда и разработчики должны подвергнуть рецензированию спецификацию системных требований, архитектуру, план тестирования системы, код оснастки системного тестирования и все дополнительные артефакты кода системного уровня. При рецензировании как отдельных сервисов, так и системы самые эффективные рецензии имеют структурную природу и назначенные роли (модератор, владелец, аналитик), а также предлагаемые последующие действия, обеспечивающие применение рекомендаций в системе. Как минимум команда должна провести неформальное рецензирование, в ходе которого эти артефакты будут рассматриваться с одним или несколькими коллегами. Независимо от выбранного метода рецензирование требует высокой степени взаимного участия, а также командного духа приверженности качеству. Разработка качественных программных продуктов — командная дисциплина.

Этот список неполон. Я привожу его не для того, чтобы предоставить вам доскональный перечень всех необходимых активностей по контролю качества, а скорее для того, чтобы заставить вас думать обо всем, что необходимо сделать в проекте для контроля качества.

Активности обеспечения качества

В плане проекта всегда должны присутствовать активности обеспечения качества. Обеспечение качества уже обсуждалось в предыдущих главах (особенно в главе 9). Вам следует включить в процесс и план проекта следующие активности обеспечения качества:

- **Обучение.** Если ваши разработчики не будут пытаться самостоятельно разбираться в новых технологиях, это обойдется вам значительно дешевле (а качество будет намного выше). Отправляя разработчиков на курсы повышения квалификации (или устраивая обучение на территории организации), вы моментально исключите многие дефекты, обусловленные трудностями на начальном этапе обучения или нехваткой опыта.
- **Определение ключевых стандартных процедур.** Разработка программного обеспечения — область настолько сложная и нетривиальная, что в ней ничто не должно оставаться на волю случая. Если у вас нет стандартных процедур (SOP, Standard Operating Procedures) для всех ключевых активностей, выделите время на то, чтобы выработать и записать их.
- **Принятие стандартов.** Наряду со стандартными процедурами также необходимо иметь стандарт проектирования (см. приложение B) и стандарт программирования. Соблюдение правил предотвратит проблемы и дефекты.
- **Участие в обеспечении качества.** Активно общайтесь со специалистом, занимающимся контролем качества. Предложите этому специалисту проанализировать процесс разработки, внести изменения для обеспечения качества и создать процесс, который будет одновременно эффективным и понятным. Этот процесс должен обеспечивать своевременное выявление и устранение корневых причин дефектов или еще лучше — предотвращать само возникновение проблем.
- **Сбор и анализ ключевых метрик.** Метрики позволяют обнаруживать проблемы до их возникновения. К их числу относятся метрики, относящиеся к разработке (точность оценок, эффективность, количество обнаруженных дефектов при рецензировании, тренды качества и сложности), а также метрики времени выполнения (время работоспособности и надежность). При необходимости включите активности для построения программных средств для сбора метрик; учитывайте косвенные затраты на сбор и анализ метрик на регулярной основе. Подкрепите практику стандартной процедурой, запрещающей дальнейшие действия при аномальных метриках.
- **Подведение итогов.** Как описано в предыдущем размере, проводите частичное подведение итогов в процессе работы и подведите итоги проекта в целом после его завершения.

Качество и культура

Многие руководители не доверяют своим командам. Они слишком часто сталкивались с разочарованиями и не видят особой корреляции между усилиями команды и желаемыми результатами. Такие руководители склонны управлять всем в ручном режиме. Это прямой результат хронического дефицита доверия. Разработчики реагируют на ручное управление раздражением и апатией и те-

ряют все оставшиеся крохи ответственности. Ситуация продолжает деградировать, а негативное мнение руководителей только подтверждается.

Лучший способ преодолеть эту динамику — заразить команду неустанным стремлением к качеству. При полной приверженности качеству команда будет направлять каждую активность с точки зрения качества, восстанавливая разрушенную культуру и создавая атмосферу высоких технических стандартов. Чтобы добиться такого состояния, необходимо сформировать правильный контекст и окружение. На практике это означает «делать все, о чем написано в этой книге» — и многое другое.

Результатом будет переход от ручного управления к обеспечению качества. Доверие к людям, которые должны сами контролировать качество своей работы, — суть делегирования полномочий. Когда это будет сделано, вы узнаете, что качество является самым эффективным методом управления проектом, поскольку оно требует минимального руководства и при этом обеспечивает максимальную производительность команды. Руководители направляют усилия на то, чтобы сформировать правильные условия работы для команды; они доверяют команде и ожидают, что она выпустит безупречную программную систему в рамках сроков и бюджета.

Приложения

A Отслеживание проекта

Одну из самых превратно понятых цитат приписывают фельдмаршалу Хельмуту фон Мольтке: «Ни один план сражения не переживает встречи с противником». С тех пор это утверждение вырывалось из контекста и воспринималось как оправдание для полного отсутствия планирования — что полностью противоречит его исходному намерению. Фон Мольтке, известный как архитектор франко-прусской войны 1870 года, был гением военного планирования, которому ставят в заслугу целую серию поразительных военных побед. Фон Мольтке понимал, что перед лицом стремительно изменяющихся обстоятельств успех возможен только в том случае, если вы не зависите от единственного статического плана. Необходимо проявить гибкость, чтобы быстро переключаться между несколькими тщательно продуманными вариантами. Исходный план всего лишь дает шанс на успех; он совмещает доступные ресурсы с целями настолько хорошо, насколько это возможно. С этого момента вы должны непрерывно отслеживать проект по плану и пересматривать его по мере необходимости. Часто для этого приходится создавать модификации текущего плана, переключаться на альтернативные, заранее подготовленные варианты или разрабатывать совершенно новые варианты.

В контексте проектирования системы и планирования проекта наблюдение фон Мольтке остается таким же актуальным, как и 150 лет назад. Методы планирования проектов, описанные в книге, служат двум целям. Во-первых, они помогают принять обоснованное решение в ходе анализа SDP, чтобы ответственные за принятие решений могли выбрать жизнеспособный вариант. Такой вариант становится отправной точкой — достаточно хорошей для того, чтобы проект имел шанс на успех. Вторая цель планирования проекта — адаптация плана в процессе выполнения. Менеджер проекта должен постоянно сверять происходящее с планом, а архитектору приходится применять средства планирования проектов и перепланировать проект, чтобы он соответствовал реальности. Часто такая работа принимает форму умеренных итераций перепланирования проекта. Желательно избегать любых серьезных переделок и вместо этого плавно направлять проект серией небольших поправок. Иначе

объем необходимой корректировки может оказаться неподъемным, что приведет проект к провалу.

Хороший план проекта не из тех документов, которые вы подписываете и убираете в ящик, чтобы он уже никогда не увидел дневного света. Хороший план проекта — живой документ, который постоянно пересматривается для выполнения обязательств. Для этого необходимо знать, в какой точке вы находитесь относительно плана, куда направляетесь и какие меры собираетесь предпринять в ответ на изменяющиеся обстоятельства. В этом заключается вся суть отслеживания проекта.

Отслеживание проекта относится к управлению и исполнению и не входит в круг обязанностей программного архитектора. По этой причине я включил отслеживание проекта в книгу, но только в приложение к основному обсуждению проектирования систем и планирования проектов.

Жизненный цикл и статус активностей

Для планирования проекта необходимо определить текущее состояние проекта в отношении различных ресурсов и активностей. В предыдущих главах при обсуждении активностей проекта активности рассматривались как атомарные единицы, при этом каждой активности ставилась в соответствие продолжительность или оценка затрат. Такой подход позволяет планировать проект независимо от того, что происходит внутри активностей. Для отслеживания проектов такой подход недостаточен. Каждую активность в проекте, будь то сервис или активность, не связанная с программированием, можно преобразовать в отдельный жизненный цикл с внутренними задачами. Такие задачи могут быть последовательными, чередующимися на временной оси или итеративными. Например, на рис. А.1 показан возможный жизненный цикл сервиса.

Каждый сервис начинается со спецификации требований сервиса (SRS, Service Requirement Spec). Спецификация SRS может состоять всего из нескольких абзацев или страниц с описанием того, что должен делать сервис. Архитектор должен проанализировать SRS. Имея SRS, разработчик может перейти к написанию плана тестирования сервиса (STP, Service Test Plan) с перечнем всех способов, с помощью которых он позднее может продемонстрировать неработоспособность сервиса. Даже при участии старших разработчиков, способных выполнить подробное проектирование сервиса, разработчик не всегда может начать подробное проектирование без получения дополнительной информации о природе сервиса. Чтобы получить такую информацию, лучше всего проделать некоторую работу по построению; так вы получите непосредственное представление о возможностях технологии и о доступных вариантах подробного проектирования. Вооружившись полученной информацией, разработчик может перейти к проектированию подробностей сервиса, которые затем архи-

тектор может проанализировать (возможно, с другими участниками). После того как результат подробного проектирования будет утвержден, разработчик может перейти к построению кода сервиса. Параллельно с сервисом разработчик строит тестового клиента, работающий по принципу «белого ящика». Этот тестовый клиент позволяет разработчику протестировать каждый параметр, каждое условие и путь обработки ошибок вызовом отладчика для эволюционирующего кода. После того как код будет завершен, разработчик проводит его анализ с архитектором и другими разработчиками, интегрирует сервис с другими сервисами и, наконец, проводит юнит-тестирование по принципу «черного ящика» в соответствии с планом тестирования.

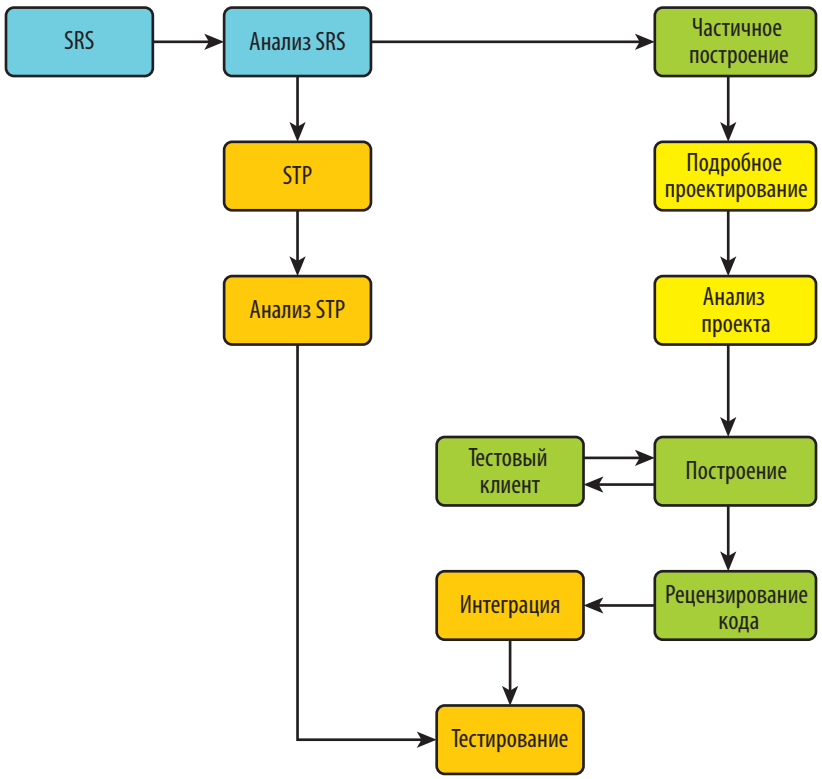


Рис. А.1. Жизненный цикл разработки сервисов

Обратите внимание: каждая задача анализа на диаграмме должна завершиться успешно. Неудача при рецензировании заставляет разработчика повторить предыдущую внутреннюю задачу. Для ясности на рис. А.1 эти повторные попытки не обозначены.

Критерий выхода из фазы

Независимо от конкретного потока операций жизненного цикла многие активности содержат внутренние фазы: *Требования*, *Подробное проектирование*, *Построение* и т. д. Каждая фаза состоит из одной или нескольких внутренних задач, как показано на рис. А.2.

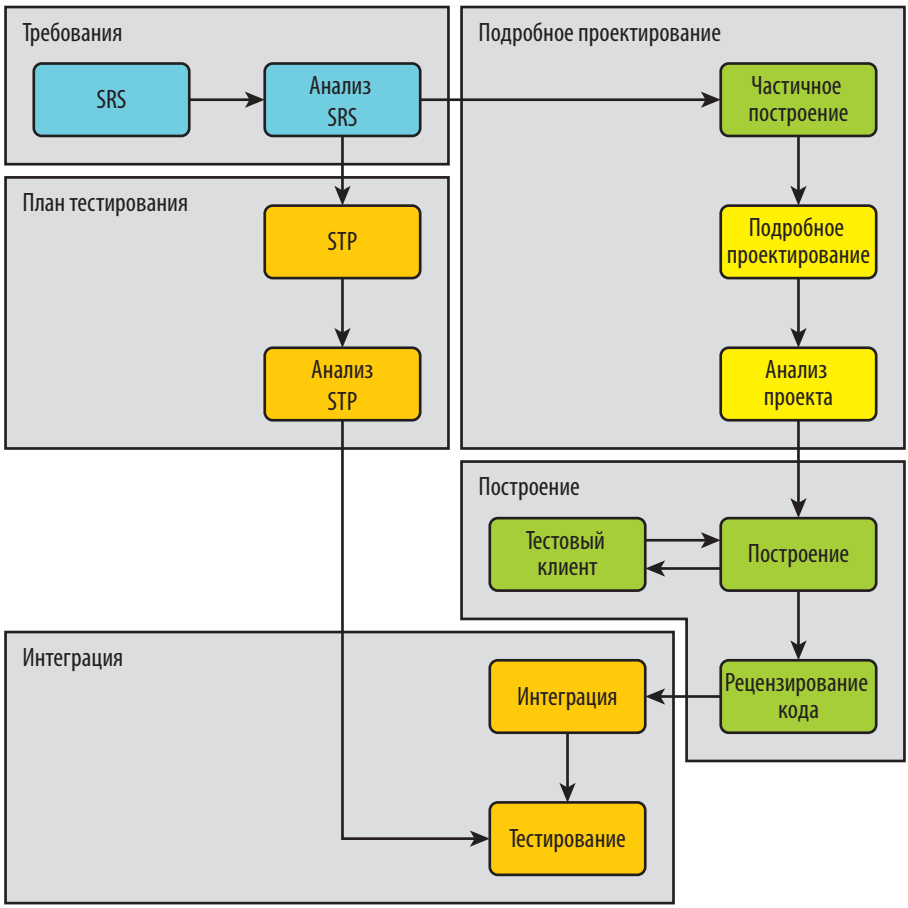


Рис. А.2. Фазы активности и задачи

Например, фаза *Подробного проектирования* может включать построение, собственно подробное проектирование и анализ проектировочного решения. Фаза *Построения* может включать фактическое построение, тестового клиента и анализ кода.

Для поддержки отслеживания важно определить бинарный критерий выхода для каждой фазы, то есть для принятия решения о том, завершена фаза или нет, используется единственное условие. С жизненным циклом на рис. А.2 можно использовать анализ и тестирование как бинарные критерии выхода для фазы. Например, фаза *Построение* завершается с проведением рецензирования кода, а не просто при записи кода в базу системы контроля версий.

Вес фазы

Хотя каждая активность может состоять из нескольких фаз, эти фазы могут вносить разный вклад в завершение активности. Вклад фазы может оцениваться в форме веса — в данном случае измеряемого в процентах. Например, рассмотрим активность с фазами из табл. А.1. В этом примере фаза *Требования* отвечает за 15% завершения активности, тогда как фаза *Подробное проектирование* отвечает за 20% завершения.

Веса фаз могут назначаться несколькими способами. Например, вы можете оценить важность фазы или же оценить продолжительность каждой фазы в днях и разделить на сумму продолжительностей всех фаз. Также можно делить на количество фаз (например, с 5 фазами каждая фаза отвечает за 20%) и даже учитывать тип активности. Например, можно решить, что фаза *Требования* должна иметь вес 40% для активности UI-проектирования и только 10% для активности журнала.

Таблица А.1. Фазы и веса активностей

Фаза активности	Вес (%)
Требования	15
Подробное проектирование	20
План тестирования	10
Построение	40
Интеграция	15
Итого	100

Для точного отслеживания не так важно, какой способ будет использоваться для распределения весов фаз, важно лишь, чтобы этот способ последовательно применялся для всех активностей. В большинстве серьезных проектов количество фаз по всем активностям будет исчисляться сотнями. В среднем все расхождения при назначении весов будут компенсировать друг друга.

Статус активности

С заданным бинарным критерием выхода и весом каждой фазы можно вычислить прогресс каждой активности в любой момент времени. При отслеживании прогресс определяется как статус завершения активности (или всего проекта) в процентах.

Формула прогресса активности:

$$A(t) = \sum_{j=1}^{m} W_j,$$

где:

- W_j — вес фазы j активности;
- m — количество завершенных фаз активности на время t;
- t — точка на временной оси.

Прогресс активности на момент времени t равен сумме весов всех фаз, завершенных ко времени t. Например, согласно табл. А.1, если первые три фазы (*Требования*, *Подробное проектирование* и *План тестирования*) завершены, то активность завершена на 45% (15 + 20 + 10).

По аналогии с вычислением прогресса активности можно (и нужно) отслеживать объем работы по каждой активности. В отслеживании объем работы равен величине прямых затрат на активность (или на весь проект), выраженной в процентах от оцениваемых прямых затрат для активности (или для всего проекта). Формула объема работы по активности:

$$C(t) = \frac{S(t)}{R},$$

где:

- $S(t)$ — накапливаемые прямые затраты на активность на момент времени t;
- R — оценка прямых затрат на активность;
- t — точка на временной оси.

Очень важно то, что объем работы не связан с прогрессом. Например, активность, оцениваемая с продолжительностью в 10 дней при фиксированных ресурсах, может быть завершена только на 60% через 15 дней после начала. Затраты по этой активности уже достигли 150% запланированных прямых затрат.

> **ПРИМЕЧАНИЕ** Как прогресс, так и объем работы являются безразмерными величинами: их значения выражаются в процентах. Это позволяет избежать конкретных величин и сравнивать их в одном анализе.

Статус проекта

Формула для прогресса проекта:

$$P(t) = \frac{\sum_{i=1}^{N}(E_i \times A_i(t))}{\sum_{i=1}^{N} E_i},$$

где:

- E_i — оценка продолжительности активности i;
- $A_i(t)$ — прогресс активности i на момент времени t;
- t — момент времени;
- N — количество активностей в проекте.

Общий прогресс проекта на момент времени t вычисляется как отношение двух сумм оценок. Первая — сумма всех оценок продолжительности всех отдельных активностей, умноженных на прогресс активностей. Вторая — сумма всех оценок активностей. Следует заметить, что эта простая формула определяет прогресс проекта по всем активностям, разработчикам, жизненным циклам и фазам.

Прогресс и осваиваемая ценность

В главе 7 рассматривается концепция осваиваемой ценности. Формула планируемой осваиваемой ценности как функция времени и формула прогресса проекта очень похожи. Если все активности завершаются точно так, как было запланировано, то прогресс по времени будет совпадать с планируемой осваиваемой ценностью, пологой S-образной кривой проекта. Прогресс проекта просто равен фактической осваиваемой ценности на заданную дату.

Чтобы продемонстрировать это обстоятельство, рассмотрим простой проект в табл. А.2. Предположим, на момент времени t UI-активность была завершена всего на 45%. Так как 45% от 20% соответствуют 9%, работа, выполненная до настоящего момента в UI-активности, вносит вклад 9% к завершению проекта. Аналогичным образом вычисляется фактическая осваиваемая ценность по всем активностям проекта в момент времени t.

Суммирование фактической осваиваемой ценности по всем активностям в табл. А.2 показывает, что на момент времени t проект завершен на 40,25%. Это же значение будет получено по формуле прогресса:

$$P(t) = \frac{40 \times 1{,}0 + 30 \times 0{,}75 + 40 \times 0{,}45}{40 + 30 + 40 + 20 + 40 + 30} = 0{,}4025.$$

Таблица А.2. Название таблицы

Активность	Продолжительность	Ценность (%)	Завершено (%)	Фактическая осваиваемая ценность (%)
Начальная стадия	40	20	100	20
Сервис Доступ	30	15	75	11,25
UI	40	20	45	9
Сервис Менеджер	20	10	0	0
Вспомогательные средства	40	20	0	0
Тестирование системы	30	15	0	0
Итого	200	100	–	40,25

Накапливаемый объем работы

Формула объема работы по проекту:

$$D(t) = \frac{\sum_{i=1}^{N}(R_i \times C_i(t))}{\sum_{i=1}^{N} R_i} = \frac{\sum_{i=1}^{N}(R_i \times \frac{S_i(t)}{R_i})}{\sum_{i=1}^{N} R_i} = \frac{\sum_{i=1}^{N} S_i(t)}{\sum_{i=1}^{N} R_i},$$

где:

- R_i — оценка прямых затрат активности i;
- $C_i(t)$ — объем работы для активности i в момент времени t;
- $S_i(t)$ — накапливаемые прямые затраты по активности i в момент времени t;
- t — точка на временной оси;
- N — количество активностей в проекте.

Общий объем работы по проекту — это сумма прямых затрат по всем активностям, разделенная на сумму всех оценок прямых затрат по всем активностям. Таким образом, объем работы определяется как доля расходования общих прямых затрат от запланированных прямых затрат по проекту.

Еще раз подчеркнем сходство объема работ по проекту с формулой планируемой осваиваемой ценности. Если на каждую активность назначается один ресурс, если затраты по каждой активности в конечном итоге точно совпадают с запланированными и активности завершаются к запланированным

датам, кривая объема работ будет соответствовать кривой планируемой осваиваемой ценности. Если на активность планируется более (или менее) одного ресурса, вам придется отслеживать объем работ по отдельной кривой планируемой осваиваемой ценности. Тем не менее в большинстве проектов две кривые должны быть очень похожими. Для простоты в оставшейся части приложения предполагается, что на каждую активность планируется один ресурс.

Накапливаемые косвенные затраты

Косвенные затраты по проекту в основном являются функцией времени и структуры команды; они не зависят от объема работ или прогресса отдельных активностей. Вы можете воспользоваться методом, сходным с описанным ранее, для определения текущего статуса косвенных затрат. Необходимо выявить участников команды, вносящих основной вклад в косвенные затраты (таких, как основная команда, DevOps или тестировщики), и отслеживать время, потраченное на проект, за вычетом прямых затрат (если они есть).

Так как косвенные затраты не зависят ни от прогресса, ни от объема работ по проекту, отслеживание косвенных затрат не принесет особой пользы. Скорее всего, вы получите восходящую прямую линию, которая не будет предполагать никаких мер по исправлению ситуации.

Тем не менее отслеживание косвенных затрат принесет пользу в одной ситуации: в отчетах по общим затратам проекта на заданный момент, для чего косвенные затраты должны суммироваться с прямыми. В оставшейся части этой главы при отслеживании проекта и его сравнении с планом учитываются только накапливаемые прямые затраты (объем работ).

Отслеживание прогресса и объема работ

Объединение фактического прогресса проекта с объемом работ позволяет вам определить текущий статус проекта. Эти вычисления следует повторять периодически. Это позволит вам построить график прогресса и объема работ в отношении планируемой осваиваемой ценности проекта. На рис. А.3 продемонстрирована эта форма отслеживания для нашего примера.

Синяя линия на рис. А.3 изображает планируемую осваиваемая ценность проекта. Планируемая осваиваемая ценность должна иметь вид пологой S-образной кривой; вскоре вы увидите, почему в данном примере она отклоняется от этой формы. До момента времени, показанного на графике, зеленая линия изображает реальный прогресс проекта (фактическая осваиваемая ценность), а красная — трудозатраты.

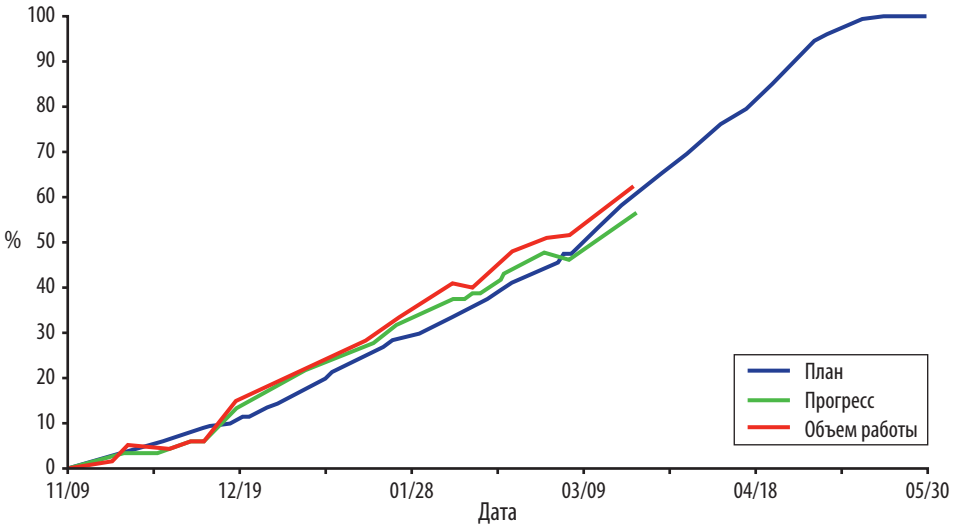


Рис. А.3. Пример отслеживания проекта

Прогнозирование

Отслеживание проекта позволяет точно узнать, в каком состоянии находится проект и в каком состоянии он находился ранее. Однако настоящий вопрос заключается не в том, каков текущий статус проекта, а скорее в том, в каком направлении он идет. Чтобы ответить на этот вопрос, можно воспользоваться кривыми прогресса и объема работы. Взгляните на обобщенное представление проекта на рис. А.4.

Для простоты на рис. А.4 пологие S-образные кривые заменены их линиями тренда линейной регрессии, обозначенными сплошными линиями на диаграмме. Синяя линия представляет запланированную осваиваемую ценность проекта. В идеале линия прогресса и линия объема работы должны совпадать с линией плана. Предполагается, что проект завершится при достижении планируемой осваиваемой ценности на 100% (точка 1 на рис. А.4). Тем не менее из диаграммы видно, что зеленая линия (реальный прогресс) проходит ниже плана.

Экстраполируя зеленую линию прогресса, мы получаем пунктирную зеленую линию на рис. А.4. Из диаграммы видно, что к моменту достижения точки 1 линия прогресса достигает всего 65% завершения (точка 2 на рис. А.4). Проект будет завершен, когда линия прогресса достигнет 100%, или точки 3 на рис. А.4. Время точки 3 представлено точкой 4 на рис. А.4, а разность между точками 4 и 1 определяет прогнозируемое превышение срока.

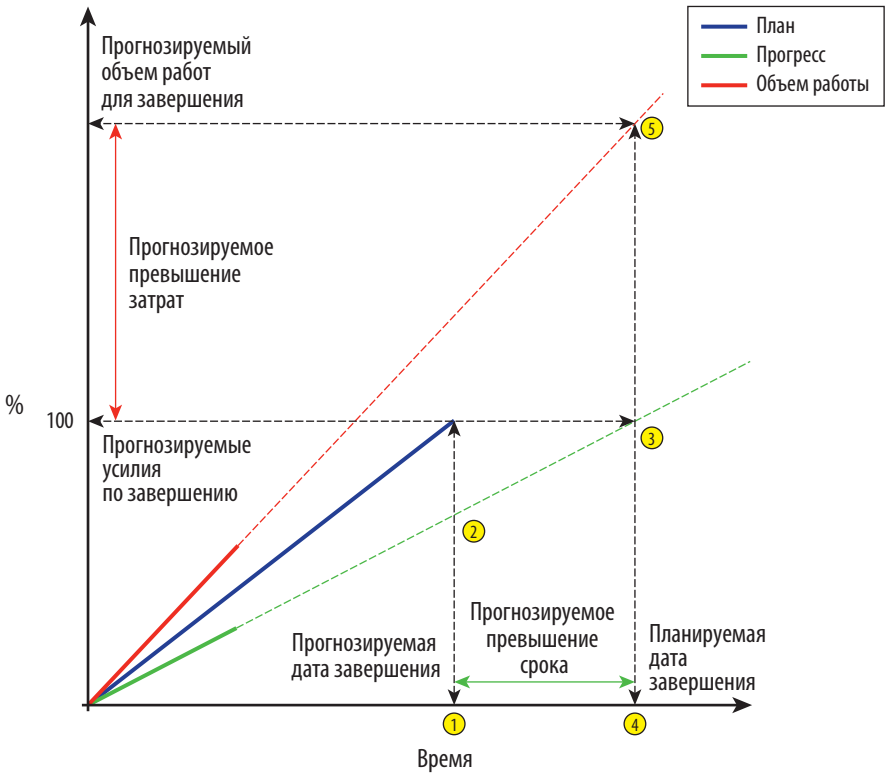


Рис. А.4. Проекции прогресса и объема работ

Аналогичным образом можно спроецировать линию объема работ и найти точку 5 на рис. А.4. Разность в объемах работ между точками 5 и 3 на рис. А.4 определяет прогнозируемое превышение прямых затрат (в процентах) для проекта.

> **ПРИМЕЧАНИЕ** Так как косвенные затраты обычно линейны по времени, прогнозируемое превышение срока в процентах также обозначает прогнозируемое превышение косвенных затрат.

Предположим, проект занимает год, и вы измеряете показатели с недельными интервалами. Через месяц у вас появятся четыре опорные точки — этого достаточно для построения регрессионной линии тренда, которая хорошо аппроксимирует измеренные значения прогресса и объема работы. Как говорилось в главе 7, угол наклона кривой осваиваемой ценности представляет результативность команды. Следовательно, уже через месяц ведения проекта, рассчитанного на год, вы достаточно хорошо представляете, в каком направлении движется проект; для этого вы используете прогноз, сверенный с фактической

результативностью команды. Исходная планируемая осваиваемая ценность была всего лишь отправной точкой. Прогнозируемые линии прогресса и объема работы с большей вероятностью соответствуют тому, что произойдет на самом деле. На рис. А.5 изображены прогнозы для рис. А.3. В соответствии с ними можно ожидать, что проект превысит срок приблизительно на месяц (или 13%) при 8% превышения по объему работы.

На рис. А.3 и рис. А.5 планируемая осваиваемая ценность представляет собой усеченную пологую S-образную кривую, потому что отслеживание для этого проекта началось после анализа SDP. Намеренная потеря очень пологой начальной части плана позволяет обеспечить лучшую аппроксимацию кривых линиями тренда.

> **ПРИМЕЧАНИЕ** В файлах, прилагаемых к книге, содержится электронная таблица, которая позволяет легко отслеживать прогресс проекта и автоматически строить прогнозы по линиям тренда.

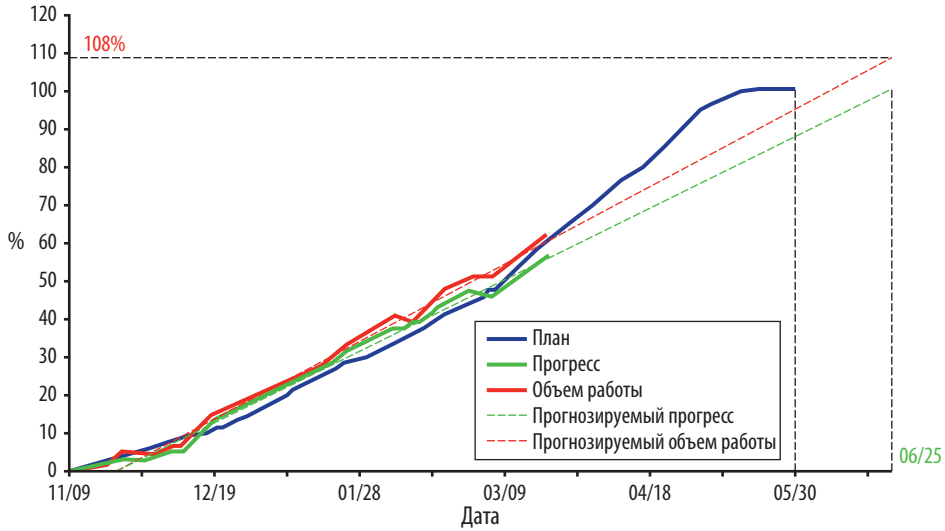


Рис. А.5. Пример прогнозирования прогресса и объема работ

Прогнозирование и корректировка

Прогнозирование прогресса и объема работ предоставляет непревзойденную возможность увидеть, где находится проект и куда он направляется. Тогда появляется возможность снова поднять планку и обсудить возможные меры по исправлению ситуации. Помните, что при возникновении проблем важно исправлять саму проблему, а не ее симптомы. Например, нарушение сроков

и превышение запланированных трудозатрат — только симптомы, а не сама проблема. В этом разделе перечислены основные симптомы, с которыми вы можете столкнуться, возможные меры по исправлению и даже рекомендации по выбору оптимального способа действий.

Все хорошо

Возьмем прогнозы прогресса и объема работ на рис. А.6. На диаграмме линии прогнозируемого прогресса и объема работ совпадают с планом, а проект ориентирован на выполнение своих обязательств. С текущим состоянием ничего делать не нужно; незачем помогать или пытаться что-то улучшить. Знать, когда нужно что-то сделать, важно — но не менее важно знать, когда не нужно делать ничего.

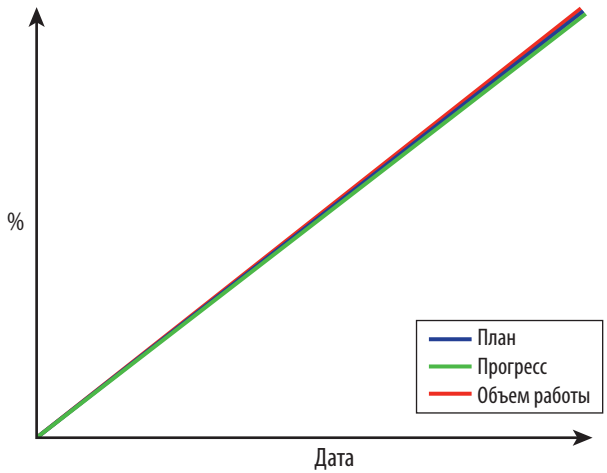


Рис. А.6. В проекте все хорошо

Соблюдение плана

Обеспечение согласования прогресса и объема работ с планом в такой степени должно быть естественным состоянием дел в любом проекте, потому что только так можно выполнить обязательства. Большинство людей неправильно понимают, что требуется для соблюдения сроков. Многие полагают, что в ходе работы над проектом они могут отклоняться от своих обязательств, а потом какими-то героическими усилиями и решимостью добьются соблюдения сроков в конце. И хотя это может оказаться правдой, шансы на такой исход невелики, и конечно, рассчитывать на него не стоит. Во многих проектах нет героев, и такие проекты не переживут слишком резких колебаний.

Основополагающее правило управления проектами выглядит так:

> *Единственный способ выдержать срок в конце — не отставать во время проекта.*

Нахождение в пределах исходного (или даже пересмотренного) плана никогда не происходит само по себе. Оно требует постоянного отслеживания со стороны менеджера проекта и многочисленных корректировок в ходе выполнения. Вы должны реагировать на информацию, раскрываемую траекторией прогнозов, и по возможности предотвращать возникновение разрывов между прогрессом, объемом работ и планом.

Заниженная оценка

Возьмем прогнозы осваиваемой ценности и объема работ на рис. А.7. Очевидно, у этого проекта большие проблемы. Прогресс отстает от плана, а объем работ превышает его. Наиболее вероятное объяснение состоит в том, что вы недооценили проект и его активности.

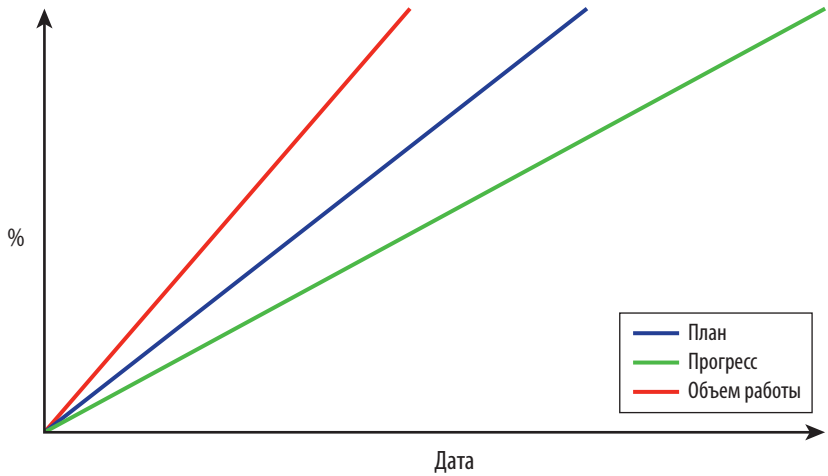


Рис. А.7. Прогнозы, указывающие на заниженную оценку

Меры по исправлению ситуации

В случае с заниженной оценкой возможны две очевидные меры по исправлению ситуации. Первая — пересмотреть оценки и повысить их на основании (теперь известной) результативности команды. Вы видите, когда прогнозируемая линия прогресса достигнет 100%, и эта точка становится новой датой заверше-

ния проекта. Фактически вы сдвигаете вниз линию плана до тех пор, пока она не встретится с линией прогресса.

Это типичное решение в проектах, зависящих от функциональности, в которых вы должны добиться паритета с продуктом конкурента или унаследованной системой, а выпуск системы, в которой отсутствуют ключевые аспекты, не имеет смысла.

Тем не менее смещение срока не подойдет для проекта, который должен быть выпущен к заданной дате. В этом случае следует использовать исправительные меры второго типа: сокращение масштаба проекта. При сокращении масштаба осваиваемая ценность, произведенная командой до настоящего момента, вносит больший вклад в проект, а линия прогресса поднимется до линии плана.

Конечно, можно применить некоторое сочетание смещения срока и сокращения масштаба, а прогнозирование прогресса поможет определить, в какой именно пропорции должна применяться та или иная мера. Выбранная вами реакция определит дальнейшее перепланирование проекта.

К сожалению, инстинктивная реакция многих руководителей, которые не желают мириться с нарушением сроков или сокращением масштаба, — привлечение к проекту дополнительных людей. Как заметил Фредерик Брукс в книге «Мифический человеко-месяц», это равносильно попытке тушить пожар бензином.

Есть несколько причин, по которым добавление людей в запаздывающий проект почти всегда только ухудшает ситуацию. Во-первых, даже если добавление людей приблизит линию прогресса к линии плана, линия объема работы при этом взлетит вверх. Нет смысла исправлять один аспект проекта за счет нарушения другого (особенно если в проекте уже используется больше людей, чем было запланировано, как показано на рис. А.7). Во-вторых, новых людей необходимо ввести в курс дела и обучить. Для этого необходимо прервать работу других участников команды, которые часто являются самыми квалифицированными работниками и, что еще важнее, с большой вероятностью работают на критическом пути: остановка или замедление их работы будет означать дальнейшую задержку проекта. В конечном итоге вы будете оплачивать как время адаптации, так и время, потерянное существующей командой, помогающей с адаптацией. Наконец, даже без затрат на адаптацию новая команда увеличится в размерах, а следовательно, станет менее эффективной.

У этого правила существует одно исключение; оно встречается в самом начале проекта. В начале проекта можно вложиться в массовую адаптацию участников команды. Что еще важнее, в начале проекта добавление новых людей может быть приемлемым, потому что вы можете переключиться на более агрессивный и уплотненный вариант планирования проекта. Такое решение обычно требует дополнительных ресурсов из-за параллельной работы.

Следует учитывать, что уплотнение проекта приведет к повышению уровня риска и сложности, поэтому вы должны тщательно взвесить все последствия нового решения.

Утечка ресурсов

Рассмотрим прогнозы прогресса и объема работы на рис. А.8. В этом проекте как прогресс, так и объем работы лежат ниже линии плана, а прогресс расположен даже ниже линии объема работы. Часто такая ситуация возникает из-за утечки ресурсов: люди назначаются на ваш проект, но при этом отвлекаются на работу над другим проектом. В результате они не могут уделять проекту необходимое время, и прогресс снова замедляется. Утечка ресурсов является обыденным явлением в программной отрасли. Мне доводилось наблюдать утечки, составлявшие до 50% от объема работы.

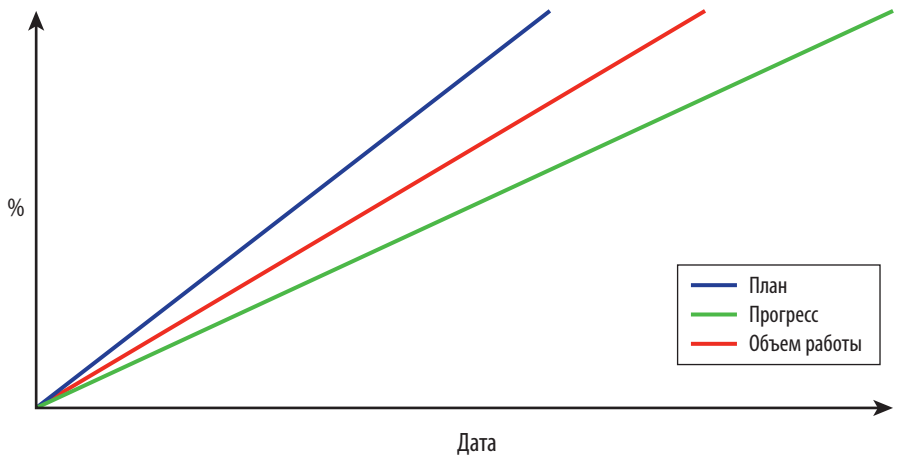


Рис. А.8. Прогнозы, указывающие на утечку ресурсов

Меры по исправлению ситуации

При выявлении утечки ресурсов возникает инстинктивное желание попросту закрыть ее. Тем не менее закрытие утечки нередко приводит к негативным последствиям: оно может подорвать выполнение других проектов, причем вина будет возложена на вас. В такой ситуации лучше всего устроить встречу с участием менеджера проекта, в который утекают ресурсы вашей команды, вас и руководителя низшего уровня, отвечающего за оба проекта. После представления диаграммы с прогнозами (вроде рис. А.8) вы предлагаете руководителю два варианта. Если другой проект важнее вашего, то зеленая линия на рис. А.8 представляет то, что ваша команда может сделать в этих новых обстоятельствах, и дедлайн не-

обходимо сдвинуть с учетом этих обстоятельств. Но если ваш проект важнее, то менеджер проекта из другой команды должен немедленно закрыть доступ к системе контроля версий участникам вашей команды и, возможно, даже назначить некоторые лучшие ресурсы из другого проекта в ваш проект для компенсации уже причиненного ущерба. Если варианты решения будут представлены подобным образом, независимо от решения руководства вы только выиграете и сохраните возможность выполнения своих обязательств.

Завышенная оценка

Рассмотрим прогнозы прогресса и объема работы на рис. А.9. Хотя может показаться, что в проекте дела идут очень хорошо, потому что линия прогресса лежит выше линии плана, в реальности проект находится под угрозой из-за завышенной оценки. Как обсуждалось в главе 7, завышенная оценка так же опасна, как и заниженная. Дополнительная проблема с проектом на рис. А.9 заключается в том, что проект тратит больше усилий, чем того требует план. Это может быть связано с тем, что на проект было назначено слишком много народа, или с тем, что работа над проектом ведется в незапланированном параллельном режиме.

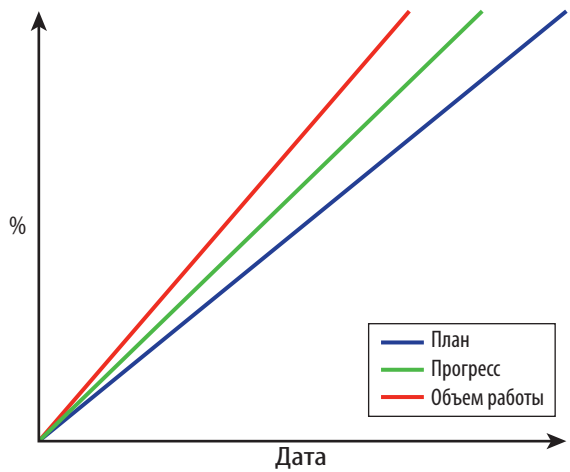


Рис. А.9. Прогнозы, указывающие на завышенную оценку

Меры по исправлению ситуации

Простая мера для исправления ситуации при завышенной оценке — пересмотр оценок со снижением и введение нового срока. Синяя линия плана на рис. А.9 поднимается до соприкосновения с линией прогресса; степень ее наклона можно расчитать. К сожалению, приближение дедлайна с большой вероятностью

будет иметь только отрицательные последствия. Часто выпуск системы с опережением графика не имеет никаких преимуществ. Например, заказчик может не оплатить работу до согласованного срока, серверы могут быть не готовы или команде будет нечем заняться после этого. В то же время сокращение продолжительности проекта повышает давление на команду. Люди по-разному реагируют на давление. Умеренное давление может иметь положительные результаты, тогда как избыточное давление лишает мотивации. Если участники команды пасуют перед давлением, проект разваливается. Обычно бывает трудно понять, где именно проходит эта тонкая грань.

Другая возможная мера — сохранение дедлайна с заниженными оценками и расширением масштаба проекта. Добавление задач (возможно, начало работы над следующей подсистемой) сократит фактическую осваиваемую ценность, а линия прогресса опускается до синей линии плана на рис. А.9. Добавление ценности всегда полезно, но оно несет с собой риск избыточного давления.

Лучшее решение проблемы с завышенной оценкой — освобождение части ресурсов. При этом красная линия объема работ опускается, так как команда меньшего размера обходится дешевле. Линия прогресса опускается, потому что у меньшей команды результативность сокращается. Меньшая команда также должна быть более эффективной. Если завышенная оценка будет обнаружена на достаточно ранней стадии, вы даже можете выбрать решение с меньшим уплотнением.

Подробнее о прогнозах

Прогнозы позволяют проанализировать, куда направляется проект, задолго до обострения проблем. Снова взгляните на рис. А.4. Если вы будете ждать, пока проект достигнет точки 2 на диаграмме, а затем корректировать его до синей линии, это потребует болезненных, если не разрушительных, действий. При помощи прогнозирования можно выявлять тренды на более ранней стадии, и выполнять небольшие корректировки до того, как между линиями возникнет серьезный разрыв. Чем раньше выполнены действия, тем больше времени останется для их вступления в силу, тем менее разрушительными они будут для остальных частей проекта, тем проще ему будет получить одобрение руководства и тем выше вероятность успеха. Всегда лучше действовать на опережение, чем реагировать постфактум, и унция профилактики часто стоит фунта лекарства.

Как и при вождении машины, в процессе реализации проекта вносится множество мелких изменений вместо нескольких крупных. Хорошие проекты всегда идут плавно — по планируемой осваиваемой ценности, диаграмме распределения комплектования или, как в данном случае, по графикам прогресса или объема работы.

Учтите, что продемонстрированные приемы анализируют тренды проекта, а не его фактическое состояние. Это правильный механизм управления проектом. Вернемся к аналогии с машиной: когда вы ведете машину вперед, вы не смотрите только на асфальт или только в зеркало заднего вида. Не так важно, где находится машина в данный момент и где она была ранее. Когда вы ведете машину, вы смотрите прежде всего туда, где она должна оказаться, и принимаете меры для выполнения этого прогноза.

Неконтролируемое расширение масштаба проекта

Как ни странно, руководство может даже пытаться изменять масштаб проекта без изменения продолжительности и ресурсов, назначенных на проект. В свою очередь, это создает проблему с выполнением обязательств.

Объединение прогнозирования с планированием проекта — наилучший способ преодоления непредвиденных изменений в масштабе проекта. Когда кто-то пытается увеличить (или уменьшить) масштаб проекта и обращается к вам за одобрением или согласием, вы должны вежливо объяснить, что вернетесь с ответом чуть позже. Теперь необходимо перепланировать проект, чтобы оценить последствия изменений. Перепланирование может быть незначительным, если оно не влияет на критический путь или затраты и лежит в пределах возможностей команды. Используйте прогнозирование для оценки вашей способности реализовать проект в рамках нового плана с точки зрения фактической результативности и затрат. Конечно, изменения могут повысить продолжительность проекта, величину затрат и потребность в ресурсах. Возможно, вам также придется выбрать другой вариант планирования проекта или даже разработать совершенно новые варианты планирования.

Вернувшись к руководству, представьте новую продолжительность и общие затраты, которых требуют предложенные изменения, включая новые прогнозы, и спросите, хотят ли они этого. Если новые сроки или новые затраты оказываются неприемлемыми, значит, ничего не изменяется. Если же они будут приняты, то у вас появляются новые обязательства по срокам и затратам. В любом случае вы всегда сможете выполнить свои обязательства. Возможно, эти обязательства будут отличаться от исходных, с которыми начинался проект, но в любом случае инициатива по изменению плана будет исходить не от вас.

Формирование доверия

Многие команды разработчиков не справляются со своими обязательствами. У руководства нет причин доверять им, и существует масса причин для недоверия. В результате руководство выставляет невозможные сроки, прекрасно сознавая, что они будут нарушены. Как обсуждалось в главе 7, жесткие дедлайны кардинально снижают вероятность успеха, а неудача превращается в самоисполняющееся пророчество.

Отслеживание проекта — хороший способ вырваться из этого порочного круга. Делитесь своими прогнозами со всеми возможными руководителями и ответственными за принятие решений. Постоянно обозначайте текущее благополучное состояние проекта и тенденции на будущее. Демонстрируйте свою способность обнаруживать проблемы за месяцы до их возникновения. Настаивайте на принятии исправительных мер (или просто принимайте их). Все эти действия сформируют вашу репутацию как ответственного, заслуживающего доверия профессионала. Все это способствует формированию уважения и в конечном итоге — доверия. Когда вы завоюете доверие вышестоящих инстанций, они будут чаще давать возможность спокойно выполнять свою работу, что будет способствовать достижению успеха.

Б Проектирование контрактов сервисов

Часть I книги была посвящена архитектуре системы: вы узнали, как провести декомпозицию системы на компоненты и сервисы и как сформировать требуемое поведение из сервисов. Однако на этом процесс проектирования еще не закончен; вы должны продолжить процесс и спроектировать каждый сервис во всех подробностях.

Подробное проектирование — необъятная тема, заслуживающая отдельной книги. В этом приложении обсуждение подробного проектирования ограничивается важнейшим аспектом проектирования сервисов: открытым контрактом, который сервис предоставляет своим клиентам. Только после того, как контракт сервиса будет зафиксирован, вы сможете заполнить такие подробности внутреннего строения, как иерархии классов и сопутствующие паттерны проектирования. Эти внутренние подробности проектирования, как и контракты данных и параметры работы, привязаны к предметной области, а следовательно, выходят за рамки темы этой книги. Тем не менее на абстрактном уровне принципы проектирования, описанные здесь для контракта сервиса в целом, действуют даже на уровне контрактов данных и параметров.

В этом приложении показано, что даже в задачах, настолько привязанных к вашей системе, как проектирование контрактов ваших сервисов, некоторые правила и метрики проектирования выходят за рамки технологии сервиса, отраслевых областей или команд. Хотя идеи, изложенные в этом приложении, просты, они имеют глубокие последствия для подхода к разработке сервисов и структурированию работы по их построению.

Оценка качества проектирования

Чтобы понять, как проектировать сервисы, необходимо сначала выделить атрибуты хорошего или плохого проектирования. Возьмем системную архитектуру на рис. Б.1. Хорошо ли спроектирована эта система? В проектировании систе-

мы на рис. Б.1 используется один большой компонент, реализующий все требования к системе. Теоретически любую систему можно построить по этому принципу, с объединением всего кода в одну гигантскую функцию с сотнями аргументов и миллионами вложенных строк условного кода. Тем не менее ни один архитектор в здравом уме не скажет, что такой монстр может считаться признаком качественного проектирования. Можно сказать, что это канонический пример того, как поступать не следует. А если вспомнить главу 4, такую архитектуру также не удастся проверить.

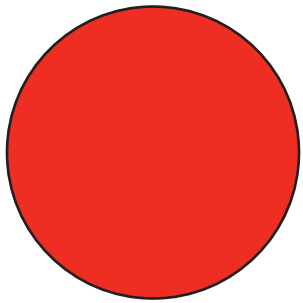


Рис. Б.1. Монолитное проектирование системы

Теперь возьмем пример архитектуры на рис. Б.2. Что вы скажете на этот раз? В архитектуре на рис. Б.2 для реализации системы используется огромное количество мелких компонентов или сервисов (для сокращения визуальной нагрузки на диаграмме не обозначены линии взаимодействия между сервисами). Теоретически любую систему можно построить подобным образом — размещением каждого требования в отдельном сервисе. И это тоже не просто плохое проектирование, а другой канонический пример того, как поступать не следует. Как и в предыдущем примере, такая архитектура также не может быть проверена.

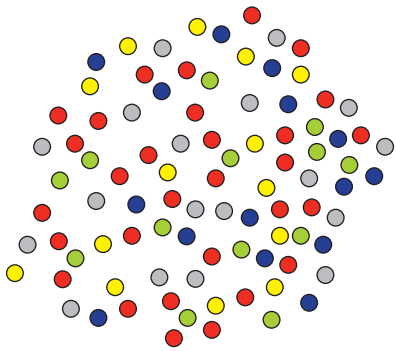


Рис. Б.2. Излишне детализированное проектирование системы

Наконец, рассмотрим проектирование системы на рис. Б.3. Можно ли сказать, что это хорошее проектировочное решение для вашей системы? Хотя вы не можете утверждать, что на рис. Б.3 изображена хорошая архитектура для вашей системы, можно утверждать, что она наверняка лучше одного огромного компонента или целой россыпи мелких компонентов.

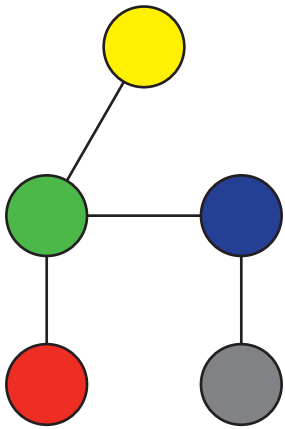


Рис. Б.3. Модульная архитектура системы

Модульность и затраты

Сама возможность определить, что архитектура системы на рис. Б.3 лучше двух предыдущих, выглядит удивительно. В конце концов, вам ничего не известно о природе системы, предметной области, разработчиках или технологии, тем не менее вы интуитивно понимаете, что она лучше. При оценке модульной архитектуры применяется ментальная модель, которую описывает рис. Б.4.

Когда система строится из меньших структурных элементов (например, сервисов), вам приходится оплачивать две составляющие затрат: затраты на построение сервисов и затраты на их объединение. Система может строиться в любой точке спектра между одним большим сервисом и бесчисленными мелкими сервисами, а на рис. Б.4 показано влияние этого решения декомпозиции на затраты по построению системы.

> **ПРИМЕЧАНИЕ** В части II этой книги затраты системы рассматриваются как функция времени и планирования проекта. На рис. Б.4 представлено еще одно измерение — как затраты системы являются функцией архитектуры системы и детализации сервисов. Разные архитектуры имеют разные кривые «время-затраты-риск».

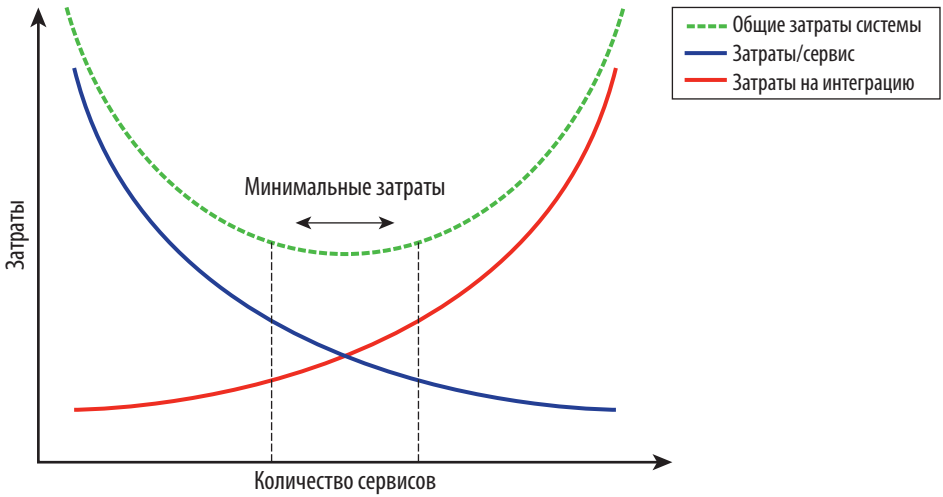


Рис. Б.4. Влияние размера и количества сервисов на затраты [изображение адаптировано по материалам Juval Lowy, Programming .NET Components, 2nd ed. (O'Reilly Media, 2003); Juval Lowy, Programming WCF Services, 1st ed. (O'Reilly Media, 2007) и Edward Yourdon and Larry Constantine, Structured Design (Prentice-Hall, 1979)]

Затраты на сервис

Реализационные затраты на сервис (синяя линия на рис. Б.4) представляют нелинейное поведение. С сокращением количества сервисов их размер увеличивается (до одного монолита в левой части кривой). Проблема в том, что с ростом сервиса его сложность возрастает в нелинейной зависимости. Вдвое больший сервис может быть в 4 раза более сложным, а вчетверо более сложный — в 20 или 100 раз более сложным. В свою очередь, повышение сложности сервиса вызывает нелинейный рост затрат. В результате сложность становится нелинейной монотонно возрастающей функцией размера. Соответственно, с сокращением количества сервисов размер сервиса увеличивается, а с каждым возрастанием размера затраты увеличиваются в нелинейной зависимости. И наоборот, в архитектуре системы с множеством сервисов (правая часть рис. Б.4) затраты на сервис стремятся к нулю.

Затраты на интеграцию

Затраты на интеграцию сервисов нелинейно зависят от количества сервисов. Это тоже является результатом сложности — в данном случае сложности возможных взаимодействий. Рост количества сервисов подразумевает большее количество возможных взаимодействий, которое добавляет сложности. Как

упоминалось в главе 12, из-за связности и каскадных эффектов с ростом количества сервисов (n) сложность растет пропорционально n^2, но даже может достигать порядка n^n. Сложность взаимодействий напрямую влияет на затраты на интеграцию; это объясняет, почему затраты на интеграцию (красная линия на рис. Б.4) также являются нелинейной кривой. Соответственно, в правой части рис. Б.4 затраты на интеграцию еще сильнее увеличиваются с ростом количества сервисов. И наоборот, в левой части кривой, где может находиться всего один огромный сервис, затраты на интеграцию падают до нуля, потому что интегрировать нечего.

Область минимальных затрат

В любой системе всегда приходится оплачивать обе составляющие затрат (затраты на реализацию и затраты на интеграцию). Пунктирная зеленая линия на рис. Б.4 представляет сумму этих двух составляющих затрат, или *общие затраты* системы. Для любой системы существует область минимальных затрат, в которой сервисы не слишком велики и не слишком малы, их не слишком мало и не слишком много. Каждый раз, когда вы проектируете систему, ее необходимо вывести в область минимальных затрат (и удерживать ее там). Обратите внимание: не обязательно находиться в самом минимуме кривой общих затрат, а нужно находиться лишь в области минимальных затрат, в которой график общих затрат системы относительно горизонтален. Когда кривая начинает выходить на горизонтальный уровень, затраты на поиск абсолютного минимума превысят экономию на системных затратах. Как упоминалось в главе 4, у каждого действия по проектированию всегда существует точка снижения эффективности, в которой оно просто становится «достаточно хорошим».

Необходимо избегать крайних точек диаграммы, потому что они нелинейно ухудшаются и становятся во много раз (и даже в десятки раз) более затратными. Проблема построения систем с нелинейно возрастающими затратами заключается в том, что средства, имеющиеся в распоряжении у организаций, по своей сути линейны. Организация может предоставить вам еще одного разработчика, потом еще одного или еще один месяц, а потом еще один месяц. Но если проблема имеет нелинейную природу, вы никогда ее не догоните. Системы, спроектированные за пределами области минимальных затрат, обречены на неудачу еще до того, как будет написана первая строка кода.

> **ПРИМЕЧАНИЕ** Решения с функциональной декомпозицией всегда оказываются на нелинейных границах рис. Б.4. Как показано в главе 2, функциональная декомпозиция приводит либо к стремительному размножению мелких областей функциональности, либо к нескольким массивным скоплениям функциональности, причем иногда даже одновременно (см. рис. 2.2).

Как объяснялось в главе 4, хорошая декомпозиция на основе нестабильности предоставляет наименьший набор структурных элементов, которые можно

объединить для выполнения всех требований — известных и неизвестных, настоящих и будущих. Такая декомпозиция обеспечивает численность сервисов в области минимальных затрат, но это ничего не говорит об их структуре. Даже когда декомпозиция следует рекомендациям Метода, нахождение сервисов в области минимальных затрат требует правильного проектирования контракта каждого сервиса.

Сервисы и контракты

Каждый сервис в системе предоставляет своим клиентам некоторый контракт. Контракт представляет собой набор операций, которые могут вызываться клиентами. Таким образом, контракт является открытым интерфейсом, предоставляемым сервисом миру. Во многих языках программирования для определения контрактов сервисов даже используется ключевое слово `interface`. Хотя контракт сервиса является интерфейсом, не все интерфейсы являются контрактами сервисов. Контракт сервиса — формальный интерфейс, который сервис обязуется поддерживать без изменений.

Воспользуемся аналогией из повседневной жизни: наше существование полно контрактов, как формальных, так и неформальных. Контракт работника определяет (часто с использованием юридических формулировок) обязательства как работодателя, так и работника друг перед другом. Коммерческий контракт между двумя компаниями определяет их взаимодействия как поставщика и потребителя некоторых услуг. Все они являются формальными формами взаимодействий, и стороны контракта часто сталкиваются с серьезными последствиями нарушения контракта или изменения его условий. С другой стороны, садясь в такси, вы вступаете в неформальный контракт: водитель обязуется безопасно довезти вас до точки назначения, а вы — оплатить его услуги. Ни один из вас не подписывает формальный контракт с описанием природы этого взаимодействия.

Контракты как грани

Контракт выходит за рамки формального интерфейса: он представляет *грань* (facet) поддерживающей его сущности для окружающего мира. Например, человек может подписать контракт найма, в котором он будет представлен как работник. У этого человека могут быть и другие грани, но работодателю видна только эта конкретная грань, и именно она его интересует. Человек может участвовать в других контрактах: договоре аренды земельного участка, брачном контракте, договоре ипотеки и т. д. Каждый из этих контрактов представляет определенную грань человека: работник, землевладелец, супруг или домовладелец. Аналогичным образом сервис может поддерживать более одного контракта.

От проектирования сервиса к проектированию контрактов

Хорошо спроектированные сервисы лежат в области минимальных затрат на рис. Б.4. К сожалению, ответить на вопрос о том, что должно считаться хорошим сервисом в этой области, достаточно сложно. Можно разве что пройти через серию разумных сверток и прийти к вопросу, на который вы можете ответить. Первая свертка предполагает однозначное соответствие между сервисами и их контрактами. С учетом этого предположения можно заменить на рис. Б.4 метку «Сервис» меткой «Контракт», а все остальное поведение диаграммы останется неизменным.

На практике один сервис может поддерживать несколько контрактов, а отдельный контракт может поддерживаться несколькими сервисами. В таких случаях кривые на рис. Б.4 смещаются влево или вправо, вверх или вниз, но их поведение остается неизменным.

Атрибуты хороших контрактов

В предположении, что между сервисами и контрактами существует однозначное соответствие, вопрос «Что такое хороший сервис?» превращается в вопрос «Что такое хороший контракт?». Хорошие контракты представляют логически последовательные, связные и независимые грани сервиса. Эти атрибуты лучше всего объяснить аналогиями из повседневной жизни.

Станете ли вы подписывать контракт с работодателем, в котором указано, что вы можете работать в компании только при условии, что проживаете по определенному адресу? Вероятно, вы откажетесь от такого контракта, потому что он логически непоследователен — ваш статус работника привязывается к вашему адресу. В конце концов, если вы выполняете согласованную работу по ожидаемым стандартам, совершенно неважно, где вы живете. Хорошие контракты всегда логически последовательны.

Станете ли вы подписывать контракт с работодателем, в котором не указано, сколько вы будете получать? Вероятно, вы откажетесь. Хорошие контракты всегда обладают связностью (cohesive) и содержат все аспекты, необходимые для описания взаимодействия, — не более и не менее.

Включите ли вы в свой брачный контракт зависимость от трудового контракта? Вероятно, вы откажетесь от такого контракта, потому что независимость контракта не менее важна. Каждый контракт (или грань) существует сам по себе и работает независимо от других контрактов.

Эти атрибуты также направляют процесс получения контракта. Захотите ли вы платить адвокату по недвижимости, чтобы он составил контракт на съем квартиры? Или просто найдете в интернете шаблон типичного контракта,

распечатаете первый результат поиска, заполните поля с адресом и размером арендной платы и удовлетворитесь этим? Если типичный контракт достаточно хорош для миллионов других арендаторов без привязки к конкретной квартире, то почему он не будет достаточно хорош для вас? Контракт необходимо дополнить так, чтобы он включал всю необходимую информацию (например, стоимость аренды). Также он должен быть независимым от других контрактов, — то есть быть по-настоящему автономной гранью.

Также обратите внимание на то, что вам не нужно пытаться найти шаблон контракта лучше того, который используется всеми остальными. Желательно просто повторно использовать тот самый контракт, который используется всеми остальными. Этот контракт настолько хорош именно из-за того, что он так хорошо подходит для повторного использования. Остается упомянуть о том, что логически последовательные, связные и независимые контракты являются повторно используемыми.

Учтите, что возможность повторного использования не является бинарным («есть/нет») свойством контракта. Каждый контракт лежит где-то в пределах спектра повторного использования. Чем более пригоден контракт для повторного использования, тем более он обладает свойствами логической последовательности, связности и независимости. Представьте контракт для сервиса на рис. Б.1. Это огромный контракт, очень сильно специализированный для одного конкретного сервиса. Безусловно, он логически непоследователен, потому что это разбухшая свалка для всего, что делает система. Вероятность того, что кто-то другой когда-либо воспользуется этим контрактом, близка к нулю.

Теперь представьте контракт одного из крошечных сервисов на рис. Б.2. Этот контракт чрезвычайно мал и очень сильно специализирован для своего контекста. Такие мелкие сущности не могут быть связными. И снова вероятность того, что кто-либо повторно воспользуется таким контрактом, близка к нулю.

Сервисы на рис. Б.3 дают некоторую надежду. Возможно, контракты сервисов на рис. Б.3 эволюционировали так, чтобы в них было включено все относящееся к их взаимодействиям — не более и не менее. Меньшее количество взаимодействий также указывает на независимость граней. Такие контракты вполне могут быть пригодными для повторного использования.

Контракты как элементы повторного использования

Важно отметить, что базовым элементом повторного использования является контракт, а не сам сервис. Например, компьютерная мышь, которой я пользуюсь для подготовки текста книги, отличается от других моделей мышей. Ее отдельные части не являются повторно используемыми. Пластиковый корпус мыши был спроектирован специально для этой модели, и его не удастся

установить на другую мышь (кроме другого экземпляра той же модели) без дорогостоящих модификаций. Тем не менее интерфейс «мышь-рука» является повторно используемым; я могу работать с мышью, и вы тоже сможете. Ваша мышь поддерживает точно такой же интерфейс; другими словами, она повторно использует этот интерфейс. Существуют многие тысячи разных моделей мышей, однако именно тот факт, что все модели используют одинаковый интерфейс, является главным признаком хорошего интерфейса. Собственно, интерфейс «мышь-рука» можно назвать интерфейсом «инструмент-рука» (рис. Б.5).

Рис. Б.5. Повторное использование интерфейсов [по мотивам Matt Ridley, The Rational Optimist: How Prosperity Evolves (HarperCollins, 2010). Изображения: Mountainpix/Shutterstock; New Africa/Shutterstock]

Наш биологический вид повторно использует интерфейс «инструмент-рука» с доисторических времен. Хотя ни один кусочек каменного топора не может повторно использоваться в мыши и ни одна электронная схема из мыши не может повторно использоваться в каменном топоре, оба инструмента используют один интерфейс. Хорошие интерфейсы являются повторно используемыми, а их сервисы — никогда.

Построение контрактов

При проектировании контрактов для ваших сервисов всегда следует мыслить категориями повторного использования. Только так можно гарантировать, что даже после построения архитектуры и декомпозиции ваши сервисы останутся в области минимальных затрат. Заметим, что стремление проектировать контракты, пригодные для повторного использования, не имеет никакого отношения к тому, будет ли кто-нибудь фактически повторно использовать ваши

контракты. Степень реального повторного использования или спрос на контракт от других сторон абсолютно несущественны. Вы должны проектировать контракты так, словно они будут повторно использоваться бесчисленное количество раз в разных системах — как в вашей текущей системе, так и в системах ваших конкурентов.

Пример проектирования

Предположим, вы хотите реализовать программную систему для управления кассовым аппаратом. Вероятно, в требованиях к такой системе будут предусмотрены сценарии использования для поиска цены товара, интеграции с системой складского учета, приема платежей, отслеживания программ лояльности покупателей и т. д. Все это легко можно сделать при помощи Метода и соответствующих *Менеджеров*, *Ядер* и т. д. Для демонстрационных целей предположим, что система должна связываться со сканером штрихкодов и читать с него идентификатор элемента. С точки зрения системы устройство для сканирования штрихкодов представляет собой обычный *Ресурс*, поэтому вы должны спроектировать контракты для соответствующих сервисов *Доступ к ресурсу*. Основные требования к сервису сканера штрихкодов: он должен уметь сканировать код элемента, регулировать ширину сканирующего луча и управлять коммуникационным портом сканера (открытие/закрытие порта). Контракт сервиса `IScannerAccess` может быть определен следующим образом:

```
interface IScannerAccess
{
   long ScanCode();
   void AdjustBeam();
   void OpenPort();
   void ClosePort();
}
```

Контракт сервиса `IScannerAccess` поддерживает необходимые функции сканера. Он легко позволяет использовать разных провайдеров сервисов (например, `BarcodeService` и `QRCodeService`) для реализации контракта `IScannerAccess`:

```
class BarcodeScanner : IScannerAccess
{...}
class QRCodeScanner : IScannerAccess
{...}
```

Возможно, вы сочтете свою задачу выполненной, потому что контракт сервиса `IScannerAccess` используется несколькими сервисами.

Нисходящий рефакторинг

Через какое-то время заказчик обращается к вам с проблемой: в некоторых ситуациях для ввода кода товара удобнее использовать другие устройства, например цифровую клавиатуру. Тем не менее контракт `IScannerAccess` предполагает, что устройство использует некоторую разновидность оптического сканера. А значит, он не позволяет управлять неоптическими устройствами вроде цифровых клавиатур или устройств чтения RFID-меток. С точки зрения повторного использования лучше абстрагировать фактический механизм чтения и переименовать операцию сканирования в операцию чтения. В конце концов, для системы не так важно, какой именно механизм будет использоваться для чтения кода товара. Также следует переименовать контракт, присвоив ему имя `IReaderAccess`, и проследить за тем, чтобы в контракте не было ничего, что бы мешало его повторному использованию другими устройствами чтения кодов. Например, операция регулировки луча `AdjustBeam()` не имеет смысла для цифровой клавиатуры. Лучше разбить исходный контракт `IScannerAccess` на два контракта и отделить нежелательную операцию:

```
interface IReaderAccess
{
   long ReadCode();
   void OpenPort();
   void ClosePort();
}
interface IScannerAccess : IReaderAccess
{
   void AdjustBeam();
}
```

Это делает возможным правильно организовать повторное использование `IReaderAccess`:

```
class BarcodeScanner : IScannerAccess
{...}
class QRCodeScanner : IScannerAccess
{...}
class KeypadReader : IReaderAccess
{...}
class RFIDReader : IReaderAccess
{...}
```

Горизонтальный рефакторинг

После внесения этого изменения проходит еще некоторое время, и заказчик решает, что программа также должна управлять лентой для подачи това-

ров на кассе. Для этого программа должна запускать и останавливать ленту, а также управлять ее коммуникационным портом. Хотя лента использует такой же порт, как и устройства чтения, она не может повторно использовать `IReaderAccess`, потому что контракт не поддерживает ленту, а лента не умеет читать коды. Более того, существует длинный список таких периферийных устройств, каждое из которых обладает собственной функциональностью, и включение каждого из них приведет к дублированию частей других контрактов. Следует заметить, что каждое изменение в бизнес-области вызовет сопутствующее изменение в предметной области системы. Это верный признак плохой архитектуры: хорошая архитектура должна быть устойчивой к изменениям в бизнес-области.

Истинная проблема заключается в том, что контракт `IReaderAccess` плохо спроектирован. Несмотря на то что все операции могут поддерживаться устройством чтения, операция `ReadCode()` не связана логически с `OpenPort()` и `ClosePort()`. В операции чтения задействована одна грань устройства (поставщик кодов), критичная для коммерческой деятельности заказчика (атомарная бизнес-операция), тогда как управление портом требует другой грани, отражающей роль сущности как коммуникационного устройства. В этом отношении интерфейс `IReaderAccess` не является логически последовательным: это просто сборная солянка для всех требований к сервису. `IReaderAccess` напоминает архитектуру на рис. Б.1 более чем что-либо другое.

Правильнее будет выполнить горизонтальный рефакторинг, выделив операции `OpenPort()` и `ClosePort()` в отдельный контракт `ICommunicationDevice`:

```
interface ICommunicationDevice
{
   void OpenPort();
   void ClosePort();
}
interface IReaderAccess
{
   long ReadCode();
}
```

Реализующие сервисы должны будут поддерживать оба контракта:

```
class BarcodeScanner : IScannerAccess,ICommunicationDevice
{...}
```

Обратите внимание: суммарная работа внутри `BarcodeScanner` не отличается от работы в исходном варианте `IScannerAccess`. Но поскольку грань коммуникации независима от грани чтения, другие сущности (например, ленты) могут повторно использовать контракт сервиса `ICommunicationDevice` и поддерживать его:

```
interface IBeltAccess
{
   void Start();
   void Stop();
}
class ConveyerBelt : IBeltAccess,ICommunicationDevice
{...}
```

Эта архитектура позволяет отделить аспект управления коммуникациями от реального типа устройства (будь то сканер штрихкодов или лента). Настоящая проблема с кассовыми аппаратами заключалась не в специфике устройств чтения, а в нестабильности типов устройств, подключенных к системе. Ваша архитектура должна базироваться на декомпозиции на основе нестабильности. Как показывает этот простой пример, данный принцип также распространяется на проектирование контрактов отдельных сервисов.

Восходящий рефакторинг

Выделение операций в отдельные контракты (как, например, выделение `ICommunicationDevice` из `IReaderAccess`) обычно применяется при наличии слабых логических связей между операциями в контракте.

Иногда идентичные операции присутствуют в нескольких несвязанных контрактах, при этом эти операции логически связаны с соответствующими контрактами. Если не включить их, это ухудшит связность контрактов. Например, представьте, что по соображениям безопасности система должна немедленно прервать работу всех устройств. Кроме того, все устройства должны поддерживать некоторую диагностику, которая гарантирует их работу в безопасных пределах. С логической точки зрения отмена является такой же операцией сканера, как и чтение, и такой же операцией подвижной ленты, как начало или остановка.

В таких случаях можно провести восходящий рефакторинг сервиса в иерархию контрактов вместо отдельных контрактов:

```
interface IDeviceControl
{
   void Abort();
   long RunDiagnostics();
}
interface IReaderAccess : IDeviceControl
{...}
interface IBeltAccess : IDeviceControl
{...}
```

Метрики проектирования контрактов

Три метода проектирования контрактов (нисходящий рефакторинг в производный контракт, горизонтальный рефакторинг в новый контракт, восходящий рефакторинг в базовый контракт) приводят к созданию оптимизированных, более компактных и пригодных к повторному использованию контрактов. Безусловно, расширение возможности повторного использования контрактов полезно, а уменьшение контрактов необходимо в том случае, когда вы начинаете проектирование с разбухших контрактов. Тем не менее слишком хорошо — тоже нехорошо. Если переусердствовать с применением этих методов, то в итоге вы получите контракты, слишком детализированные и фрагментированные, как на рис. Б.2. Следовательно, необходимо сохранять баланс между двумя противодействующими факторами: затратами на реализацию контрактов сервисов и затратами на их объединение. Метрики проектирования помогут вам выдержать баланс этих факторов.

Метрики контрактов

Вычисляя различные метрики, можно упорядочить контракты от худших к лучшим. Например, можно измерить цикломатическую сложность кода. Вряд ли вам удастся построить простую реализацию большого сложного контракта, а сложность излишне детализированных контрактов будет просто чудовищной. Также можно измерить количество дефектов в сервисах: низкокачественные сервисы с большой вероятностью являются результатом сложности плохих контрактов. Можно измерить, сколько раз каждый контракт повторно используется в системе и сколько раз контракт был изменен: очевидно, контракт, который повторно используется повсюду и не изменяется, может считаться хорошим. Метрикам можно назначить веса и ранжировать результаты. Я годами проводил такие измерения для разных технологических стеков, систем, отраслей и команд. Несмотря на все разнообразие, были обнаружены некоторые общие метрики, которые могут пригодиться для оценки качества контрактов.

Метрики размера

Контракты сервисов всего с одной операцией возможны, но их следует избегать. Контракт сервиса является гранью сущности, однако эта грань должна быть довольно примитивной, если ее можно выразить всего одной операцией. Проанализируйте эту операцию и задайте себе ряд вопросов о ней. Использует ли она слишком много параметров? Может, она недостаточно детализирована и одну операцию стоит разложить на несколько операций? Не стоит ли выделить эту операцию в существующий контракт сервиса? А может, ее лучше

разместить в следующей подсистеме, которую вы собираетесь построить? Я не могу сказать, какие именно меры следует принять, но точно скажу, что всего одна операция в контракте — плохой признак и такой контракт стоит проанализировать повнимательнее.

Оптимальное количество операций на контракт сервиса лежит в диапазоне от 3 до 5. Если вы спроектировали контракт сервиса с большим количеством операций (например, от 6 до 9), это все еще относительно неплохо, но вы начали отклоняться от области минимальных затрат на рис. Б.4. Присмотритесь к операциям и определите, нельзя ли свернуть какие-либо из них в другие операции, чтобы избежать чрезмерной детализации. Если сервис содержит одну и более операций, то с очень большой вероятностью это является признаком плохой архитектуры.

Ищите возможности выделения таких операций либо в отдельные контракты сервисов, либо в иерархии контрактов. Немедленно отклоняйте контракты с 20 и более операциями — ни в каких обстоятельствах такие контракты пользы не принесут. Такой контракт наверняка прикрывает какую-то серьезную ошибку проектирования. Большие контракты должны быть неприемлемы из-за их нелинейного влияния на затраты на разработку и сопровождение.

Интересно, что в реальном мире метрики размера контракта всегда используются для оценки качества контракта. Например, станете ли вы подписывать контракт с работодателем, состоящий всего из одного предложения? Нет, вы отклоните такой контракт, потому что одно предложение (и даже один абзац) не может отразить всех аспектов вашей роли работника. В таком контракте наверняка будут упущены важные подробности (например, условия завершения или ответственность), и он может включать положения других контрактов, незнакомых вам. С другой стороны, станете ли вы подписывать контракт на 2000 страниц? Вы даже не станете читать его, что бы он ни обещал. Даже 20-страничный контракт станет причиной для беспокойства: если характер работы требует, чтобы вы прочитали столько страниц, скорее всего, такой контракт слишком сложен и запутан. Но если контракт состоит из 3–5 страниц, вы сможете внимательно прочитать его даже в том случае, если не захотите подписывать. С точки зрения повторного использования ваш работодатель с большой вероятностью выдаст вам такой же контракт, как у всех остальных работников. Любой отход от полного повторного использования должен настораживать.

Избегайте свойств

Многие стеки разработки сервисов намеренно не включают семантику свойств в определения контрактов, хотя это ограничение легко обходится созданием операций, имитирующих свойства:

```
string GetName();
string SetName();
```

В контексте контрактов сервисов следует избегать свойств и операций, имитирующих свойства. Свойства подразумевают наличие состояния и подробности реализации. Когда сервис предоставляет доступ к свойствам, клиент знает о таких подробностях, и при изменении сервиса клиент (или клиенты) должен измениться вместе с ним. Не стоит заставлять клиента пользоваться свойствами и даже знать о них. Хорошие контракты сервисов позволяют клиентам вызывать абстрактные операции, не беспокоясь об их фактической реализации. Клиент просто вызывает операцию, а дальше сервис должен беспокоиться о том, как поддерживать свое состояние.

Хорошее взаимодействие между поставщиком и потребителем сервиса всегда происходит на уровне поведения. Такие взаимодействия должны выражаться именами вида `СделатьНечто()` — например, `Abort()`. Как именно сервис будет выполнять свою работу, клиента интересовать не должно. Такой подход тоже повторяет реальную жизнь: всегда лучше приказывать, чем просить.

Отказ от использования свойств также является хорошей практикой в любой распределенной системе. Всегда лучше хранить данные там, где они находятся, и только активизировать для них нужные операции.

Ограничение количества контрактов

Сервис не должен поддерживать более одного или двух контрактов. Поскольку контракты являются независимыми гранями сервисов, если сервис поддерживает три и более независимых бизнес-аспекта, это наводит на мысль о том, что сервис слишком велик.

Интересно, что рекомендуемое количество контрактов на сервис выводится методами оценки из главы 7. Если говорить только о порядке величины, каким должно быть количество контрактов на сервис — 1, 10, 100 или 1000? Очевидно, 100 или 1000 контрактов — признак плохого проектирования, и даже 10 контрактов многовато. Таким образом, по порядку величины количество контрактов на сервис должно быть равно 1. Применяя «метод удвоения», можно дополнительно сузить диапазон: какое количество контрактов считать более вероятным — 1, 2 или 4? Оно не может быть равно 8, потому что 8 — это почти 10, а это значение уже было исключено. Таким образом, количество контрактов на сервис лежит в диапазоне от 1 до 4. Диапазон все еще остается достаточно широким. Для сокращения неопределенности можно воспользоваться методом PERT: 1 — самая низкая оценка, 4 — самая высокая, 2 — самое вероятное число. Вычисление по формуле PERT дает результат 2,2 как число контрактов на сервис:

$$2,2 = \frac{1+4\times 2+4}{6}.$$

На практике в хорошо спроектированных системах большинство сервисов, которые мне встречались, содержали только один или два контракта, при этом

один контракт встречался чаще. Из сервисов с двумя и более контрактами дополнительные контракты почти всегда не были связаны с бизнесом; в них отражались такие аспекты, как безопасность, защита данных, хранение данных и т. д., и эти контракты повторно использовались в других сервисах.

> **ПРИМЕЧАНИЕ** Хороший способ предотвратить размножение сервисов — добавление контрактов к сервисам. Например, если ваша архитектура требует восьми Менеджеров, что превышает рекомендации из главы 3, возможно, некоторые из этих Менеджеров могут быть представлены добавлением дополнительных граней к другим Менеджерам и сокращением общего количества Менеджеров.

Использование метрик

Метрики проектирования контрактов сервисов следует рассматривать как средства оценки, а не средства проверки. Соответствие метрикам не означает, что ваша архитектура хороша — однако нарушение метрик указывает на то, что она плоха. Для примера возьмем первую версию `IScannerAccess`. Контракт этого сервиса состоит из 4 операций, прямо в диапазоне метрики от 3 до 5 операций, однако этот контракт не обладал логической последовательностью.

Старайтесь избегать проектирования, направленного на соответствие метрикам. Проектирование контракта сервиса, как и любая задача проектирования, имеет итеративную природу. Выделите время, необходимое для идентификации контракта, пригодного для повторного использования, который должен предоставляться вашим сервисом, и не беспокойтесь о метриках. Если окажется, что метрики нарушены, продолжайте работать до тех пор, пока у вас не появятся нормальные контракты. Продолжайте анализировать эволюционирующие контракты, чтобы узнать, могут ли они повторно использоваться в разных системах и проектах. Спросите себя, являются ли ваши контракты логически последовательными, связными и независимыми гранями. После того как такие контракты будут разработаны, проверьте, соответствуют ли они метрикам.

Трудности проектирования контрактов

Идеи и приемы, рассмотренные в этом приложении, прямолинейны, очевидны и просты. Проектирование контрактов является приобретаемым навыком, а практика позволяет значительно продвинуться к тому, чтобы выполнять эту работу быстро и правильно. Тем не менее между «простым» и «тривиальным» существует большая разница. Идеи, представленные в этом приложении, просты, но их не назовешь тривиальным. И действительно, жизнь полна идей простых, но при этом нетривиальных. Например, вы хотите быть здоровым. Идея простая, но она может потребовать изменений в вашей диете, стиле жизни, ежедневном режиме и даже работе — ничто из этого не является тривиальным.

Разработка контрактов сервисов, пригодных для повторного использования, является продолжительной задачей, требующей напряженных размышлений. Абсолютно необходимо сформулировать правильные контракты, иначе вы рискуете столкнуться с нелинейно худшей задачей (см. рис. Б.4). Настоящая проблема связана не с проектированием контрактов (что достаточно просто), а с тем, чтобы добиться поддержки руководства. Многие руководители не осознают последствия ошибок при проектировании контрактов. Торопясь с реализацией, они ведут проект к катастрофе. Это особенно справедливо при отсутствии старших разработчиков в команде (см. главу 14).

Даже старшим разработчикам может потребоваться обучение, чтобы они могли правильно проектировать контракты, и вы как архитектор должны направлять и обучать их. Это позволит сделать проектирование контрактов частью жизненного цикла каждого сервиса. Если команда состоит из младших разработчиков, не стоит рассчитывать на то, что они смогут выработать правильные контракты, пригодные для повторного использования, скорее у них получатся контракты, напоминающие представленные на рис. Б.1 или Б.2. Необходимо использовать подход из главы 14, чтобы либо выделить время на проектирование контрактов перед началом работы, либо (предпочтительно) привлечь нескольких опытных старших разработчиков для проектирования контрактов следующего набора сервисов параллельно с работой по построению текущего набора (см. рис. 14.6). Используйте концепции из этого приложения и рис. Б.4, чтобы объяснить вашему руководству, что действительно необходимо для построения хорошо спроектированных сервисов.

B Стандарт проектирования

Идеи, представленные в книге, просты и последовательны — как сами по себе, так и относительно всех остальных инженерных дисциплин. Тем не менее овладеть новым подходом к проектированию системы и планированию проекта может быть непросто. Со временем и при достаточной практике применение этих идей станет вашей второй натурой. Чтобы вам было проще усвоить их все, в этом приложении приводится краткий стандарт проектирования. В нем все правила проектирования из этой книги сведены в одно место в виде контрольного списка. Сам по себе список особой пользы не приносит, потому что вы все равно должны знать контекст для каждого пункта. Тем не менее обращение к стандарту поможет убедиться в том, что вы не упустили никакой важный атрибут или фактор, который необходимо принять во внимание. Стандарт играет исключительно важную роль в успешном проектировании системы и планировании проекта, поскольку он помогает применять самые эффективные методы и избегать ловушек.

Стандарт содержит элементы двух типов: директивы и рекомендации. Директива — правило, которое никогда не должно нарушаться, потому что нарушение наверняка приведет к неудаче. Рекомендация — совет, который желательно выполнять, если только у вас нет веской и нетривиальной причины для его нарушения. Нарушение рекомендации само по себе еще не означает неминуемой катастрофы, но слишком многочисленные нарушения к добру не приведут. Кроме того, если вы будете соблюдать директивы, вряд ли у вас появятся причины для нарушения рекомендаций.

Главная директива

Никогда не проектируйте под конкретные требования.

Директивы

1. Избегайте функциональной декомпозиции.
2. Выполняйте декомпозицию на основе нестабильности.
3. Применяйте компоновочное проектирование.
4. Предоставляйте функции как аспекты интеграции, а не как аспекты реализации.
5. Проектируйте итеративно, стройте инкрементно.
6. Планируйте проект для построения системы.
7. Направляйте принятие обоснованных решений, предоставляя варианты, отличающиеся по срокам, затратам и риску.
8. Стройте проект по критическому пути.
9. Не отставайте от графика во время реализации проекта.

Рекомендации по проектированию системы

1. Требования.
   1) Отражайте требуемое поведение, а не требуемую функциональность.
   2) Описывайте требуемое поведение при помощи сценариев использования.
   3) Документируйте все сценарии использования, имеющие вложенные условия, при помощи диаграмм активности.
   4) Исключайте решения, замаскированные под требования.
   5) Проверьте результат проектирования системы, убедившись в том, что оно поддерживает все основные сценарии использования.
2. Числовые характеристики.
   1) Избегайте создания более чем пяти *Менеджеров* в системе без подсистем.
   2) Старайтесь ограничиться небольшим количеством подсистем.
   3) Избегайте создания более чем трех *Менеджеров* на подсистему.
   4) Стремитесь к «золотому отношению» численности *Ядер* и *Менеджеров*.
   5) Позволяйте компонентам *Доступ к ресурсу* обращаться к нескольким ресурсам в случае необходимости.

3. Атрибуты.
   1) Нестабильность должна уменьшаться сверху вниз.
   2) Возможность повторного использования должна возрастать сверху вниз.
   3) Не инкапсулируйте изменения в природе бизнеса.
   4) *Менеджеры* должны быть почти расходными.
   5) Архитектура должна быть симметричной.
   6) Никогда не используйте открытые коммуникационные каналы для внутренних взаимодействий системы.
4. Уровни.
   1) Избегайте открытой архитектуры.
   2) Избегайте полузакрытых/полуоткрытых архитектур.
   3) Отдавайте предпочтения закрытой архитектуре.
      - Не используйте восходящие вызовы.
      - Не используйте горизонтальные вызовы (кроме очередей вызовов между *Менеджерами*).
      - Не используйте нисходящие вызовы более чем на один уровень вниз.
      - При попытках открытия архитектуры используйте очереди вызовов или асинхронную публикацию событий.
   4) Расширяйте систему за счет реализации подсистем.
5. Правила взаимодействия.
   1) Все компоненты могут обращаться с вызовами к *Вспомогательным средствам*.
   2) *Менеджеры* и *Ядра* могут обращаться с вызовами к компонентам *Доступ к ресурсу*.
   3) *Менеджеры* могут обращаться с вызовами к *Ядрам*.
   4) *Менеджеры* могут ставить в очередь вызовы к другим *Менеджерам*.
6. Запретные взаимодействия.
   1) *Клиенты* не обращаются с вызовами к нескольким *Менеджерам* в одном сценарии использования.
   2) *Менеджеры* не ставят в очередь вызовы к нескольким *Менеджерам* в одном сценарии использования.

3) *Ядра* не получают вызовы из очередей.
4) Компоненты *Доступ к ресурсу* не получают вызовы из очередей.
5) *Клиенты* не публикуют события.
6) *Ядра* не публикуют события.
7) Компоненты *Доступ к ресурсу* не публикуют события.
8) *Ресурсы* не публикуют события.
9) *Ядра*, компоненты *Доступ к ресурсу* и *Ресурсы* не подписываются на события.

Рекомендации по планированию проектов

1. Общие рекомендации.
   1) Не пытайтесь планировать механические часы.
   2) Никогда не планируйте проект без архитектуры, инкапсулирующей нестабильности.
   3) Сохраняйте и проверяйте предположения планирования.
   4) Применяйте планирование к самому процессу планирования проекта.
   5) Спланируйте для проекта несколько вариантов (как минимум нормальное, уплотненное и субкритическое решение).
   6) Общайтесь с руководством на языке Альтернативности.
   7) Всегда проводите анализ SDP перед началом основной работы.
2. Персонал.
   1) Избегайте привлечения нескольких архитекторов.
   2) Основная команда должна быть на месте с самого начала проекта.
   3) Запрашивайте минимальный уровень комплектования, необходимый для беспрепятственного продвижения по критическому пути.
   4) Всегда назначайте ресурсы с учетом временного резерва.
   5) Обеспечьте правильное распределение комплектования.
   6) Обеспечьте пологую S-образную кривую для планируемой осваиваемой ценности.
   7) Всегда назначайте компоненты разработчикам в пропорции 1:1.
   8) Стремитесь к непрерывности задач.

3. Интеграция.
   1) Избегайте точек массовой интеграции.
   2) Избегайте интеграции в конце проекта.
4. Оценки.
   1) Не завышайте оценки.
   2) Не занижайте оценки.
   3) Стремитесь к точности оценки, а не количеству разрядов.
   4) Всегда используйте 5-дневные кванты при оценках любых активностей.
   5) Оцените проект в целом, чтобы проверить решение по планированию проекта и даже начать процесс планирования.
   6) Сократите неопределенность оценок.
   7) При необходимости поддерживайте правильный стиль общения для получения оценок.
5. Сеть проекта.
   1) Рассматривайте зависимости от ресурсов как зависимости проекта.
   2) Убедитесь, что все активности находятся в цепочке, которая начинается и заканчивается на критическом пути.
   3) Убедитесь, что всем активностям назначен ресурс.
   4) Избегайте узловых диаграмм.
   5) Отдавайте предпочтение стрелочным диаграммам.
   6) Избегайте божественных активностей.
   7) Разбивайте большие проекты на сети сетей.
   8) Рассматривайте околокритические цепочки как критические.
   9) Стремитесь к низкой цикломатической сложности около 10–12.
   10) Проектируйте по уровням для сокращения сложности.
6. Время и затраты.
   1) Начинайте ускорение проекта с применения чистых и понятных практик, а не с уплотнения.
   2) Никогда не принимайте обязательства по проекту в мертвой зоне.
   3) Проводите уплотнение за счет параллельной работы, а не за счет применения лучших ресурсов.

4) Проводите уплотнение с применением лучших ресурсов осторожно и осмотрительно.
5) Избегайте уплотнения более 30%.
6) Избегайте проектов с эффективностью выше 25%.
7) Проводите уплотнение проекта, даже если вероятность выбора какого-либо из уплотненных вариантов низка.

7. Риск.
   1) Настройте диапазоны риска возникновения критичности для вашего проекта.
   2) Скорректируйте выбросы временных резервов при помощи риска активности.
   3) Проведите разуплотнение нормального решения за точкой перегиба на кривой риска.
      - При разуплотнении ориентируйтесь на риск 0,5.
      - Ориентируйтесь на точку перегиба риска, а не на конкретное значение риска.
   4) Не увлекайтесь чрезмерным разуплотнением.
   5) Проводите разуплотнение решений с планированием по уровням — возможно, агрессивное.
   6) Удерживайте нормальные решения на уровне риска менее 0,7.
   7) Избегайте риска ниже 0,3.
   8) Избегайте риска выше 0,75.
   9) Избегайте вариантов проекта более рискованных или более безопасных, чем в точках пересечения риска.

Рекомендации по отслеживанию проектов

1. Определите бинарные критерии выхода для внутренних фаз активности.
2. Назначьте последовательные веса фаз по всем активностям.
3. Еженедельно отслеживайте прогресс и объем работ.
4. Отчеты о ходе работы никогда не должны базироваться на функциональности.
5. Отчеты о ходе работы всегда должны базироваться на точках интеграции.
6. Отслеживайте временной резерв околокритических цепочек.

Рекомендации по проектированию контрактов сервисов

1. Проектируйте контракты сервисов, пригодные к повторному использованию.
2. Следите за соответствием метрикам проектирования контрактов сервисов.
   1) Избегайте контрактов с одной операцией.
   2) Старайтесь ограничивать контракты сервисов от 3 до 5 операций.
   3) По возможности не используйте контракты сервисов, содержащие более 12 операций.
   4) Отвергайте контракты сервисов, содержащие более 20 операций.
3. По возможности не используйте операции, моделирующие свойства.
4. Ограничьте количество контрактов на сервис одним или двумя.
5. Избегайте распределения работы в командах с участием только младших разработчиков, не способных участвовать в проектировании контрактов.
6. Поручайте проектирование контрактов только архитекторам или компетентным старшим разработчикам.