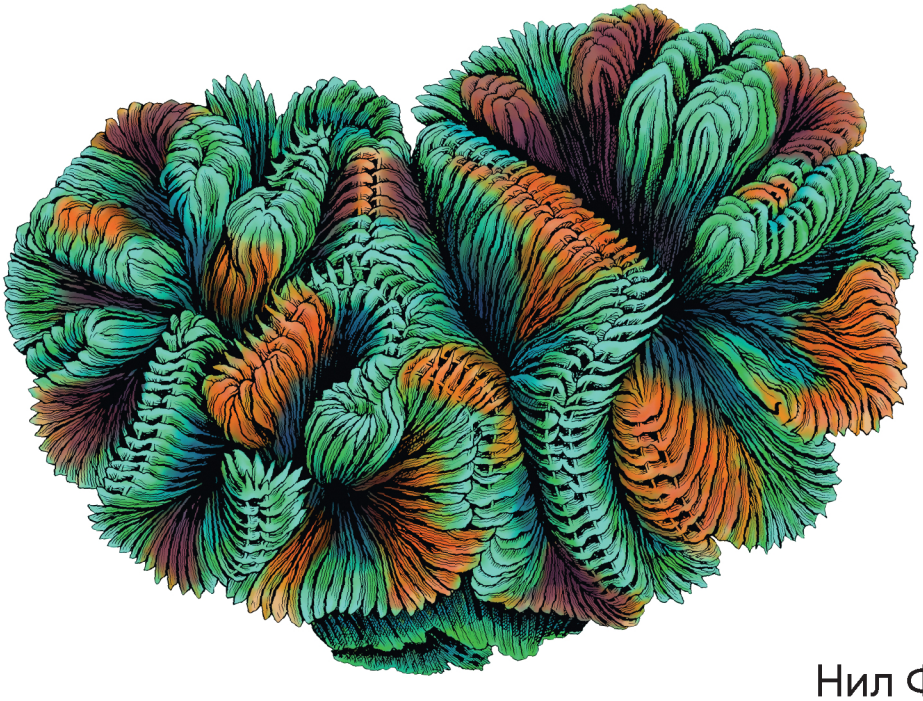
O'REILLY®

2-е издание

Эволюционная архитектура

Автоматизированное управление программным обеспечением



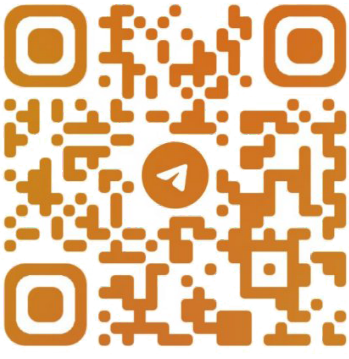
Нил Форд
Ребекка Парсонс
Патрик Куа
Прамод Садаладж

Предисловие Марка Ричардса и Мартина Фаулера

2ND EDITION

Building Evolutionary Architectures

Automated Software Governance



Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Эволюционная архитектура

Автоматизированное управление программным обеспечением

ВТОРОЕ ИЗДАНИЕ

Нил Форд, Ребекка Парсонс, Патрик Куа,
Прамод Садаладж

Оглавление

ЧАСТЬ II. СТРУКТУРА

ЧАСТЬ III. ВЛИЯНИЕ

Предисловие к первому изданию

Долгое время индустрия разработки программного обеспечения исходила из того, что архитектура продукта должна быть полностью сформирована до начала написания кода. По аналогии со строительной сферой успешной программной архитектурой считалась та, которую, подобно каркасу здания, не требовалось менять в процессе разработки: ошибки и изменения в прежние времена обходились дорого.

Эти представления были нарушены с появлением agile-подхода. Предварительное планирование архитектуры было основано на том, что требования также должны быть зафиксированы до начала написания кода, и этому принципу следовала поэтапная (или водопадная) разработка, когда вначале формулировались требования, затем создавалась архитектура, а потом начиналось строительство продукта (написание кода). Однако agile-методология поставила под сомнение само понятие фиксированных требований, декларируя, что в современном мире бизнесу необходимы регулярные изменения требований, и представила методы планирования проекта, поддерживающие внесение контролируемых изменений.

В современной agile-среде роль архитектуры часто подвергается сомнению. И, конечно, ее планирование не вписывается в новые динамичные условия. Но существует и другой подход к архитектуре, который приветствует адаптивные изменения. В рамках этого подхода архитектура выступает как сущность, требующая постоянного приложения усилий, связанных с написанием кода, чтобы реагировать как на изменение требований, так и на обратную связь, заложенную в коде. Мы назвали его *эволюционной архитектурой*, чтобы подчеркнуть, что, несмотря на то что изменения непредсказуемы, архитектура все равно может совершенствоваться в нужном направлении.

В Thoughtworks мы придерживаемся именно этого подхода. Ребекка в качестве технического директора компании руководила многими важнейшими проектами начала тысячелетия. Нил внимательно наблюдал за работой, обобщая полученные знания и передавая их другим. Патрик совмещал работу над проектами с развитием наших техлидов. Мы всегда считали архитектуру жизненно важной составляющей проекта, которую нельзя пускать на самотек. Мы делали ошибки,

но учились на них, стараясь понять, как создавать кодовые базы, эффективно приспосабливающиеся к неоднократным изменениям своего назначения.

В основе эволюционной архитектуры лежат небольшие инкрементные изменения и циклы обратной связи, дающие возможность обучаться на том, как развивается система. Развитие непрерывной доставки стало толчком к практическому применению эволюционной архитектуры. Авторы используют концепцию фитнес-функции для мониторинга состояния архитектуры. Они исследуют различные стили эволюционной архитектуры и акцентируют внимание на вопросах, связанных с долго хранимыми данными — теме, которую часто обходят вниманием. Большинство утверждений, представленных здесь, как и следует ожидать, основано на законе Конвея.

Хотя я уверен, что нам еще многое предстоит узнать об эволюции программной архитектуры, эта книга аккумулирует все текущие представления об этом подходе. Поскольку все больше людей понимают, что в XXI веке центральную роль в нашем мире играют программные системы, любой лидер нашей отрасли просто обязан знать, как эффективно реагировать на изменения, не выпадая при этом из обоймы.

— Мартин Фаулер (Martin Fowler) martinfowler.com
сентябрь 2017 г.

Предисловие ко второму изданию

Прием метафоры основан на сравнении двух разнородных понятий, подчеркивающем их основные свойства. Хороший пример — программная архитектура. Обычно ее сравнивают со структурой здания. Структурные элементы здания: внешние и внутренние стены, крыша, площадь комнат, количество этажей, даже расположение здания — можно сопоставить со структурными элементами программной архитектуры: базами данных, сервисами, протоколами связи, интерфейсами, местом развертывания (в облаке или локально) и т. д. Прежде считалось, что в обоих случаях эти единожды созданные элементы в дальнейшем с трудом поддаются изменениям. И именно здесь метафора здания перестает работать.

Сегодня параллели со строительством уже не актуальны. Хотя объяснять человеку, далекому от сферы технологий, что такое программная архитектура, попрежнему лучше всего в терминах сравнения *структуры* системы, программная архитектура должна быть способной к быстрым изменениям, в отличие от строительных конструкций. Почему же она должна быть такой податливой? Потому что современный бизнес находится в состоянии постоянных изменений — слияний, поглощений, открытия новых направлений, внедрения мер по сокращению расходов, новых организационных структур и т. д. Технологии тоже меняются, появляются новые фреймворки, технические среды, платформы и продукты. Чтобы соответствовать требованиям бизнеса и технологий, программная архитектура также должна меняться, причем такими же быстрыми темпами. Хороший пример — приобретение бизнеса крупной компанией. Помимо соответствия множеству новых бизнес-задач, архитектура основных бизнес-приложений должна быть способна масштабироваться, чтобы удовлетворить потребности увеличившейся клиентской базы, а также адаптироваться и расширяться, чтобы в нее можно было внедрить новые бизнес-функции и практики.

Многие компании знают об этом, но сталкиваются с проблемой: как сделать программную архитектуру достаточно гибкой, чтобы она выдерживала быстрые темпы изменения бизнеса и технологий? Ответ на этот вопрос дан в книге, которую вы держите в руках. Второе издание опирается на концепции управляемых инкрементных изменений, представленные в первом издании, и делится новейшими методами, знаниями и советами, касающимися фитнес-функций, автоматизации управления архитектурой и эволюционных данных. Применяя их, вы обеспечите гибкость архитектуры, достаточную, чтобы соответствовать непрерывным изменениям современного мира.

— Марк Ричардс (Mark Richards) developertoarchitect.com
октябрь 2022 г.

Введение

Когда в 2017 году мы работали над первым изданием этой книги, идея эволюционной программной архитектуры все еще казалась довольно радикальной. Во время одной из первых презентаций на эту тему Ребекку даже обвинили в непрофессионализме, поскольку она озвучила предположение, что программная архитектура *способна развиваться* с течением времени, — ведь считалось, что архитектура никогда не меняется.

Реальность учит нас, что системы должны развиваться, чтобы соответствовать новым требованиям пользователей и постоянно меняющейся экосистеме разработки.

На момент публикации первого издания существовало еще мало инструментов, подходящих для реализации описанных нами методов. К счастью, разработка программного обеспечения продолжает развиваться, и появляются все новые решения, облегчающие создание эволюционных архитектур.

Структура книги

Мы изменили структуру по сравнению с первым изданием, четко разграничив две основные темы: инженерные практики создания эволюционных программных систем и структурные подходы, которые облегчают этот процесс.

В первой части мы рассматриваем механизмы и практики разработки, подходящие для реализации эволюционной архитектуры, в том числе методы, инструменты, категории и все, что необходимо читателям для понимания этой темы.

Программная архитектура также включает *проектирование структур*, и существуют решения, которые облегчают их эволюцию (и управление ими). Мы расскажем об этом в части II, которая, кроме того, включает обзор стилей архитектуры, а также принципов проектирования, основанных на связанности, повторном использовании и других важных структурных соображениях.

В программной архитектуре практически не существует изолированных элементов; многие принципы и практики эволюционной архитектуры предполагают

комплексное взаимодействие составляющих процесса разработки, о чем мы расскажем в части III.

Практические примеры и PenultimateWidgets

В книге разбираются практические примеры. На момент работы над материалом все авторы занимались (а некоторые до сих пор занимаются) консультированием, и многие примеры, которые здесь представлены, основаны на их реальном опыте. Хотя мы не могли разглашать конфиденциальные детали, мы все же хотели разобрать несколько подходящих ситуаций, чтобы не ограничиваться абстрактной теорией. Так родилась вымышленная компания PenultimateWidgets, в которой происходят все рассматриваемые ситуации.

При поготовке второго издания мы попросили коллег привести еще более наглядные примеры применения обсуждаемых методов. Каждый такой кейс в книге представлен как случай из практики PenultimateWidgets, но все они взяты из реальных проектов.

Условные обозначения

В этой книге используются следующие условные обозначения:

Курсив

Курсивом выделены новые термины или важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова.

`Моноширинный полужирный шрифт`

Используется для команд или другого текста, который пользователь должен ввести самостоятельно.

`Моноширинный курсив`

Используется для текста, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, имен файлов и каталогов.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее примечание.

Этот рисунок указывает на предупреждение.

Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу http://evolutionaryarchitecture.com.

Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Эволюционная архитектура», 2-е издание, Нил Форд, Ребекка Парсонс, Патрик Куа и Прамод Садаладж (O'Reilly). Авторское право Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage, 2023; 978-1-492-09754-9».

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Авторы выражают огромную благодарность коллегам, которые давали подсказки и помогали найти вдохновение для многих примеров фитнес-функций, представленных в этой книге. Мы говорим спасибо (порядок имен не имеет значения) Карлу Найгарду, Александру Гедерту, Сантошкумару Паланисами, Рави Кумару Пасумарти, Индхумати В., Маноджу Б. Нараянану, Нираджу Сингху, Сирише К., Гирешу Чунчуле, Мадху Дхарваду, Венкату В., Абдулу Джелани, Сентхилу Кумару Мургешу, Мэтту Ньюману, Сяоджуну Рену, Арчане Ханал, Хайко Герину, Слину Кастро, Фернандо Тамайо, Ане Родриго, Питеру Гиллард-Моссу, Анике Вайс, Биджешу Виджаяну, Назнин Рупавалле, Кавите Миттал, Вишванату Р., Дхивье Садасивам, Рози Тейшейре, Грегорио Мело, Аманде Маттос (Carl Nygard, Alexander Goedert, Santhoshkumar Palanisamy, Ravi Kumar Pasumarthy, Indhumathi V., Manoj B. Narayanan, Neeraj Singh, Sirisha K., Gireesh Chunchula, Madhu Dharwad, Venkat V., Abdul Jeelani, Senthil Kumar Murugesh, Matt Newman, Xiaojun Ren, Archana Khanal, Heiko Gerin, Slin Castro, Fernando Tamayo, Ana Rodrigo, Peter Gillard-Moss, Anika Weiss, Bijesh Vijayan, Nazneen Rupawalla, Kavita Mittal, Viswanath R., Dhivya Sadasivam, Rosi Teixeira, Gregorio Melo, Amanda Mattos) и всем, кого мы не назвали.

Нил благодарит всех участников конференций, где он выступал в течение последних нескольких лет, помогавших оттачивать и пересматривать необходимые знания лично и особенно онлайн, в связи с глобальной пандемией. Спасибо всем сотрудникам первой линии, которые активно помогали авторам в это трудное время. Он также благодарит технических рецензентов, предоставивших ценные отзывы и советы. Кроме того, Нил благодарит своих кошек — Амадея, Фаучи и Линду Рут — за то, что отвлекали его и тем самым помогали находить новые идеи. Кошки никогда не думают о прошлом или будущем; они всегда в настоящем, поэтому время, проведенное с ними, ценно как возможность ощутить себя здесь и сейчас. Спасибо также нашему соседскому «коктейльному клубу», который задумывался как предлог повидаться с друзьями, а превратился в мозговой центр, объединивший соседей. И наконец, Нил благодарит свою бесконечно терпеливую жену, которая с улыбкой переживала сначала его командировки, а затем их внезапное отсутствие и другие неудобства, связанные с его работой.

Ребекка благодарит всех коллег, участников конференций, спикеров и авторов, которые на протяжении ряда лет предлагали идеи, инструменты и методы и подогревали интерес к эволюционной архитектуре. Она присоединяется к словам благодарности Нила в адрес научных редакторов за их внимание к деталям и комментарии. Кроме того, Ребекка благодарит соавторов за познавательные беседы и обсуждения во время совместной работы над книгой. Особую благодарность она выражает Нилу за выдающуюся дискуссию или даже спор, который

они вели несколько лет назад по поводу различий эмерджентной и эволюционной архитектур. С тех пор эти идеи прошли долгий путь.

Патрик благодарит всех своих коллег и клиентов в ThoughtWorks, которые помогли понять необходимость эволюционной архитектуры и испытать ее идеи на практике. Он также присоединяется к словам благодарности Нила и Ребекки в адрес научных редакторов, чьи отзывы очень помогли улучшить книгу. Наконец, он благодарит соавторов за проведенные вместе последние несколько лет и за возможность тесного сотрудничества, невзирая на разницу в часовых поясах и перелеты, из-за которых личные встречи были редкостью.

Прамод благодарит всех коллег и клиентов, которые служили источником вдохновения для изучения и продвижения новых идей и мышления. Он благодарит соавторов за содержательные обсуждения всех аспектов архитектуры. Он также благодарит рецензентов — Кассандру Шум (Cassandra Shum), Генри Матчена (Henry Matchen), Луку Меццалиру (Luca Mezzalira), Фила Мессенджера (Phil Messenger), Владика Кононова, Венката Субраманиума (Venkat Subramanium) и Мартина Фаулера — за вдумчивые комментарии, которые очень помогли авторам. И наконец, он благодарит своих дочерей, Арулу и Архану, за радость, которую они приносят в его жизнь, и свою жену Рупали за любовь и поддержку.

О научном редакторе русского издания

Анна Белых — старший инженер-разработчик в компании КРОК. Участвовала в проектировании и разработке высоконагруженных информационных систем разных масштабов и с применением различных архитектурных решений. Также занималась вопросами производительности серверной (бэкенд) части информационных систем.

ЧАСТЬ I

Механика

Эволюционная архитектура включает две обширные области: *механику* и *структуру*.

Механика эволюционной архитектуры описывает инженерные практики и методы проверки, благодаря которым архитектура и эволюционирует, и в этом она пересекается с управлением архитектурой. Механика включает инженерные практики, тестирование, метрики и т. д. — все, что делает возможным эволюцию программного обеспечения. В части I дается определение и приводятся многочисленные примеры механики эволюционной архитектуры.

Вторая область *эволюционной архитектуры* — структура или топология программных систем. Действительно ли определенные стили архитектуры больше подходят для создания систем, которым проще эволюционировать? Существуют ли структурные решения в архитектуре, которых следует избегать, чтобы облегчить эволюцию? Мы ответим на эти и другие вопросы в части II, посвященной структурированию архитектуры, способной эволюционировать.

Многие принципы построения эволюционных архитектур сочетают в себе вопросы механики и структуры; часть III книги называется «Влияние». Она содержит множество практических примеров, дает советы, рассказывает о паттернах и антипаттернах, а также обо всем, что необходимо знать архитекторам и командам, чтобы создавать эволюционирующие системы.

ГЛАВА 1

Архитектура эволюционных систем

Одна из актуальных задач разработки в целом и программной архитектуры в частности — создание систем, которые с возрастом не теряют своей привлекательности и эффективности. В этой книге рассматриваются два фундаментальных аспекта создания эволюционного ПО: использование эффективных инженерных практик agile-разработки и структурирование архитектуры для облегчения управления и внедрения изменений.

Читатели получат представление о том, насколько успешной является стратегия детерминированного управления изменениями архитектуры, когда существующее стремление обеспечить защиту ее характеристик объединяется с практическими методами, чтобы иметь возможность более эффективно изменять архитектуру, не нарушая ее.

Сложности создания эволюционных систем

> *Деградация данных (bit rot)*, также известная как *гниение данных*, *гниение бит*, *ветшание данных*, *распад бит* или *энтропия программного обеспечения*, — это постепенное ухудшение качества программного продукта с течением времени или снижение его отзывчивости, что в конечном итоге приводит к сбоям программы.

Разработчики уже давно бьются над созданием таких программных продуктов, которое не теряли бы своего качества с течением времени. О том, насколько сложна эта задача, свидетельствуют многочисленные профессиональные поговорки и выражения, в том числе разнообразие синонимов *bit rot*, приведенных выше. По крайней мере два фактора определяют эту борьбу: проблема контроля всех подвижных частей сложного программного продукта и динамичный характер экосистемы разработки ПО.

Современные программы состоят из тысяч или даже миллионов частей, которые можно изменять по ряду параметров. И каждое из этих изменений имеет

предсказуемые, а иногда и непредсказуемые последствия. Команды, которые пытаются управлять ими вручную, в конечном итоге не справляются с огромным количеством деталей и побочными эффектами от массы их возможных комбинаций.

Управлять многочисленными взаимодействиями программного обеспечения было бы довольно сложно в статичном контексте, но такого контекста не существует. Экосистема разработки программного обеспечения включает все инструменты, фреймворки, библиотеки и лучшие практики — весь накопленный багаж возможностей разработки на любой момент времени. Эта экосистема обладает равновесием, во многом похожим на равновесие биологических систем, которое разработчики могут исследовать и в рамках которого могут создавать продукты. Однако это равновесие *динамично* — постоянно появляется что-то новое, что нарушает баланс, пока не возникнет новое равновесие. Представьте себе курьера на моноколесе, везущего коробки: *динамика* — потому что курьер постоянно приспосабливается, чтобы оставаться в вертикальном положении, и *равновесие* — потому что он постоянно поддерживает баланс. В экосистеме разработки каждая новая инновация или практика может нарушить статус-кво, устанавливая новое равновесие. Образно выражаясь, мы продолжаем нагружать курьера на моноколесе новыми коробками, заставляя его заново искать равновесие.

Во многих отношениях архитекторы напоминают незадачливого курьера, постоянно балансирующего и адаптирующегося к меняющимся условиям. Инженерная практика непрерывной доставки (Continuous Delivery) представляет собой такой тектонический сдвиг равновесия: включение ранее изолированных функций, таких как операции, в жизненный цикл разработки ПО позволило переоценить значение *изменений*. Для создания систем предприятий больше не подходят статичные пятилетние планы, поскольку за этот срок вся вселенная разработки изменится, что сделает каждое долгосрочное решение потенциально бессмысленным.

Прорывные изменения трудно предсказать даже опытным специалистам. Распространение контейнеров с помощью таких инструментов, как Docker (https://www.docker.com/), — пример непредсказуемого изменения в отрасли. Однако мы можем проследить постепенное развитие контейнеризации. Когда-то операционные системы, серверы приложений и остальная инфраструктура представляли собой коммерческие объекты, требующие лицензирования и больших затрат. Многие архитектуры, разработанные в ту эпоху, были нацелены на эффективное совместное использование ресурсов. Постепенно ОС Linux стала приемлемой альтернативой для многих предприятий, что свело *денежные* затраты на операционные системы к нулю. Затем внедрение DevOps-практики автоматического выделения ресурсов машин с помощью таких инструментов,

как Puppet (https://puppet.com/) и Chef (https://www.chef.io/), сделала Linux *операционно* свободной. Когда экосистема стала свободной и широко используемой, оказалась неизбежной консолидация вокруг общих переносимых форматов: так появился Docker. Но контейнеризация была бы невозможна без поэтапной эволюции, результатом которой она и явилась.

Экосистема разработки ПО постоянно развивается, что приводит к появлению новых архитектурных подходов. Хотя многие разработчики считают, что решение о том, каким будет *следующий технологический прорыв*, принимается тайным обществом архитекторов на своих собраниях в башне из слоновой кости, на самом деле этот процесс гораздо более органичен. В экосистеме постоянно возникают новые функции, которые можно по-новому комбинировать с уже существующими и с другими новыми функциями для реализации возможностей. Например, рассмотрим недавний бум архитектур микросервисов. По мере роста популярности ОС с открытым исходным кодом в сочетании с непрерывной доставкой сообразительные архитекторы придумали, как создавать более масштабируемые системы, и для этих систем понадобилось название: так появились микросервисы.

ПОЧЕМУ В 2000 ГОДУ НЕ БЫЛО МИКРОСЕРВИСОВ

Представьте архитектора с машиной времени, который отправляется в 2000 год и обращается к операционному директору с новой идеей.

«У меня есть отличная новая концепция архитектуры, которая обеспечивает небывалую изоляцию каждой функции — это называется *микросервисы*; мы разработаем каждый сервис на основе бизнес-возможностей и сохраним слабую связанность».

«Отлично, — говорит директор по операциям. — Что вам для этого нужно?»

«Ну, мне понадобится примерно 50 новых компьютеров, и, конечно, 50 лицензий на новые операционки, и еще 20 компьютеров для изолированных баз данных и лицензии на них. Когда я смогу все это получить?»

«Уходите».

Хотя уже тогда микросервисы могли казаться хорошей идеей, экосистема для их поддержки отсутствовала.

Часть работы архитектора заключается в структурном проектировании решений конкретных проблем — у вас есть проблема, и вы решили, что с ней справится программа. Структурное проектирование можно разделить на две области:

предметная область (или требования) и *характеристики архитектуры*[1], как показано на рис. 1.1.

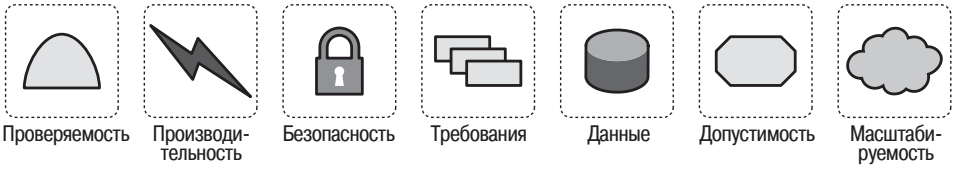


Рис. 1.1. Программная архитектура включает в себя характеристики (всевозможные понятия, заканчивающиеся на «-ость») и требования

Требования, показанные на рис. 1.1, относятся к целевой предметной области программного решения. Другие части известны как *характеристики архитектуры* (предпочтительный термин), *нефункциональные требования*, *атрибуты качества системы*, *сквозные требования* и т. д. и т. п. Независимо от названия, они представляют собой свойства, необходимые для успеха проекта — как первоначального релиза, так и долгосрочно поддерживаемого продукта. Например, такие характеристики архитектуры, как *масштабируемость* и *производительность,* имеют значение для успеха продукта на рынке, а другие, такие как *модульность,* способствуют *удобству обслуживания* системы и ее *способности эволюционировать*.

СИНОНИМЫ ТЕРМИНА «ХАРАКТЕРИСТИКИ АРХИТЕКТУРЫ»

В книге мы используем термин *«характеристики архитектуры»* для обозначения аспектов проектирования, не относящихся к предметной области. Однако во многих организациях приняты другие термины для обозначения этой концепции, в том числе «нефункциональные требования», «сквозные требования» и «атрибуты качества системы». Мы не настаиваем на выбранной терминологии — используйте свой привычный термин, если вам удобно. Это синонимичные понятия.

Программы редко бывают статичными; они продолжают развиваться по мере того, как разработчики добавляют новые функции, точки интеграции и другие изменения. Архитекторам нужны механизмы защиты характеристик архи-

[1] В литературе приняты два варианта перевода *architecture characteristics* — «характеристики архитектуры» или «свойства архитектуры». Мы остановились на первом варианте. — *Примеч. ред.*

тектуры наподобие модульных тестов, сфокусированных на характеристиках архитектуры, меняющихся с разной скоростью и иногда подверженных влиянию факторов, не связанных с предметной областью. Например, изменение базы данных может быть обусловлено выбором общих технических решений, внедряемых в компании, и не зависеть от решения для конкретной предметной области.

В этой книге описываются механизмы и методы проектирования, с которыми вы будете так же уверенно управлять архитектурой, как высокоэффективные команды управляют другими аспектами процесса разработки.

Архитектурные решения — это решения, подразумевающие наличие серьезных компромиссов для каждого возможного варианта. Для целей этой книги *архитекторами* мы называем всех, кто принимает архитектурные решения, независимо от их должности в организации. Кроме того, важные архитектурные решения практически всегда требуют взаимодействия с другими сотрудниками.

НУЖНА ЛИ АРХИТЕКТУРА AGILE-ПРОЕКТАМ?

Это популярный вопрос, который задают те, кто уже какое-то время работает по принципам agile. Agile-методология призвана избавиться от *бесполезных* накладных расходов, что не всегда означает отказ от каких-то шагов, например от проектирования. Как часто бывает в архитектуре, масштаб диктует ее уровень. Сравните со строительством: если мы строим собачью будку, нам не нужна сложная архитектура; нам нужны только материалы. Но если мы строим 50-этажное офисное здание, нам необходим проект. Точно так же, если мы разрабатываем сайт с простой базой данных, нам не нужна архитектура; мы можем собрать его из доступных компонентов. Однако чтобы спроектировать высокомасштабируемый и общедоступный веб-сайт, например для массовой продажи билетов на мероприятия, мы должны тщательно продумать множество компромиссов.

Вместо вопроса *«Нужна ли архитектура agile-проектам?»*, архитекторам нужно задать другой вопрос: при каком минимальном объеме необязательного проектирования они сохранят способность итерировать первоначальную версию проекта, чтобы создавать на ее основе более подходящие решения.

Эволюционная архитектура

Как механизмы эволюции, так и решения, которые принимают архитекторы при разработке программного обеспечения, основываются на следующем тезисе:

> Эволюционная архитектура ПО поддерживает *управляемые* и *инкрементные* изменения в *разных измерениях*.

Он включает три характеристики, которые мы более подробно рассмотрим ниже.

Управляемые изменения

Определив ключевые характеристики, команда должна *управлять* изменениями в архитектуре, чтобы они соответствовали этим характеристикам. Для этого мы заимствуем концепцию из области эволюционных вычислений, называемую *фитнес-функцией* (fitness function), или иначе *функцией пригодности (приспособленности)*. Фитнес-функция — это объективная функция, используемая для оценки того, насколько разрабатываемое решение достигает поставленных целей. В эволюционных вычислениях она определяет, становится ли алгоритм совершеннее со временем. Другими словами, по мере создания каждого варианта алгоритма функция пригодности определяет, насколько «пригодным» является этот вариант, основываясь на определении «пригодности» разработчиком.

В эволюционной архитектуре мы преследуем аналогичную цель: по мере развития архитектуры нам требуются механизмы, позволяющие оценить, как изменения влияют на важные характеристики архитектуры, и предотвратить ухудшение этих характеристик с течением времени. Подобие фитнес-функции охватывает множество механизмов, которые мы используем для предотвращения нежелательных изменений архитектуры, включая метрики, тесты и прочие инструменты проверки. Когда архитектор выделяет ту или иную характеристику архитектуры, которую он собирается защитить, он определяет одну или несколько функций пригодности для защиты этой характеристики.

Традиционно архитектура рассматривалась с позиции управления, и лишь с недавних пор архитекторы свыклись с возможностью внесения изменений в архитектуру. Фитнес-функции архитектуры позволяют принимать решения в контексте потребностей организации и задач бизнеса на видимой и проверяемой основе. Эволюционная архитектура не подразумевает разработку без ограничений и ответственности. Скорее это подход, который уравновешивает необходимость быстрых изменений и строгого соблюдения системных и архитектурных характеристик. Функция пригодности управляет принятием архитектурных решений, направляя архитектуру и позволяя вносить изменения, необходимые для поддержки меняющейся бизнес- и технологической среды.

Мы используем *фитнес-функции* как руководство по эволюции архитектур; подробно мы рассмотрим их в главе 2.

Инкрементные изменения

Определение *инкрементные* относится к двум аспектам программной архитектуры: созданию и развертыванию программных продуктов.

Что касается создания, то архитектуру, допускающую небольшие, постепенные изменения, легче усовершенствовать, поскольку разработчики могут вносить из-

менения меньшего масштаба. В части же развертывания инкрементные изменения относятся к уровню модульности и ослаблению сцепления бизнес-функций и к способам их выражения в архитектуре. Приведем пример.

Предположим, что у PenultimateWidgets, крупного продавца разных безделушек, есть каталог продуктов, созданный на основе микросервисной архитектуры и поддерживающий современные практики разработки. Одна из функций каталога позволяет пользователям оценивать продукты, ставя им звезды. Звездами можно оценивать и работу других сервисов PenultimateWidgets (службы поддержки клиентов, службы доставки и т. д.). Однажды команда разработчиков функции звездного рейтинга выпускает новую версию наряду с существующей, и теперь можно ставить ползвезды — небольшое, но важное улучшение. Остальные сервисы, использующие систему оценки, не обязаны переходить на новую версию, но могут постепенно мигрировать на нее в удобном для себя режиме. В DevOps-практику PenultimateWidgets входит архитектурный мониторинг не только сервисов, но и связей между ними. Когда группа эксплуатации замечает, что в течение определенного времени обращения к какому-то сервису отсутствуют, она автоматически исключает этот сервис из экосистемы.

Это пример инкрементных изменений на архитектурном уровне: исходный сервис может работать рядом с новым до тех пор, пока он нужен другим сервисам. Команды могут мигрировать к новым паттернам в удобном для себя ритме (или по мере необходимости), а старая версия автоматически становится мусором.

Чтобы инкрементные изменения были успешными, требуется совместное использование нескольких методов Continuous Delivery. Не всегда обязательно использовать все эти методы; скорее, они обычно естественным образом встречаются вместе. О том, как внедрять инкрементные изменения, мы поговорим в главе 3.

Разные измерения архитектуры

> Отдельных, изолированных систем не существует. Мир непрерывен. Где провести искусственную границу вокруг системы, зависит от того, какая перед нами цель — на какие вопросы надо найти ответ.
>
> *— Донелла Х. Медоуз (Donella H. Meadows)*

Древние греки учились анализировать Вселенную на основе неподвижных точек, в результате чего появилась классическая механика. Однако более точные приборы и более сложные явления сделали возможным развитие в начале XX века теории относительности. Ученые поняли, что явления, которые они раньше считали изолированными, на самом деле взаимосвязаны. С 1990-х годов про-

двинутые архитекторы все чаще рассматривают программную архитектуру как многомерную. Непрерывная доставка расширила эти представления, включив в них и операции. Однако программные архитекторы обычно ограничиваются *технической* архитектурой — тем, как компоненты программного обеспечения сочетаются друг с другом. Но это лишь одно измерение программного проекта. Если архитектор хочет создать архитектуру, которая способна развиваться, он должен учитывать *все* взаимосвязанные части системы, на которые влияют изменения. Как мы знаем из физики, что все относительно, так и архитекторы знают, что программный проект имеет несколько измерений.

Для создания программных систем, способных эволюционировать, архитекторы должны думать не только о технической архитектуре. Например, если проект включает реляционную базу данных, то структура и отношения между объектами базы данных также будут эволюционировать со временем. Точно так же не стоит создавать систему, в которой в результате развития возникнет уязвимость безопасности. Все это примеры *измерений* (dimensions) архитектуры — ее частей, которые сочетаются друг с другом, обычно пересекаясь. Некоторые измерения вписываются в понятие *характеристики архитектуры* (те самые слова на «-ость», о которых мы уже говорили), но *измерения* на самом деле шире и охватывают понятия, традиционно выходящие за рамки технической архитектуры. Каждый проект имеет свои измерения, которые архитектор должен учитывать, думая об эволюции. Вот некоторые общие измерения, которые влияют на возможность эволюции в современных программных архитектурах.

Техническое измерение

Реализационные части архитектуры: фреймворки, зависимые библиотеки и язык(и) реализации.

Данные

Схемы баз данных, компоновка таблиц, планирование оптимизации и т. д. Обычно это зона ответственности администратора базы данных.

Безопасность

Политики и рекомендации по безопасности и инструменты для выявления недостатков.

Операционное/системное измерение

Как архитектура соотносится с существующей физической и/или виртуальной инфраструктурой: серверами, кластерами машин, коммутаторами, облачными ресурсами и т. д.

Каждое из этих представлений формирует *измерение* архитектуры — целенаправленно выделенные части, поддерживающие определенное представление.

Наша концепция архитектурных измерений включает в себя традиционные архитектурные характеристики («-ости»), а также любые другие функции, применяемые для создания программного продукта. Каждая из них формирует архитектурное представление, которое нам необходимо сохранять по мере эволюции предметной области и изменения условий среды.

Мысля в терминах архитектурных измерений, архитектор получает возможность проводить анализ способности различных архитектур эволюционировать; этот анализ основан на оценке того, как каждое важное измерение реагирует на изменения. Поскольку системы со временем все теснее связываются со вступающими друг с другом в противоречие зонами ответственности (масштабируемость, безопасность, распределение, транзакции и т. д.), архитекторы должны отслеживать в проектах больше измерений. Чтобы построить систему, способную эволюционировать, архитекторы должны учесть пути ее эволюции по всем ключевым измерениям.

Архитектурная часть проекта включает требования к программному продукту и другие измерения. Для защиты этих характеристик по мере эволюции архитектуры и экосистемы во времени можно использовать фитнес-функции, как показано на рис. 1.2.

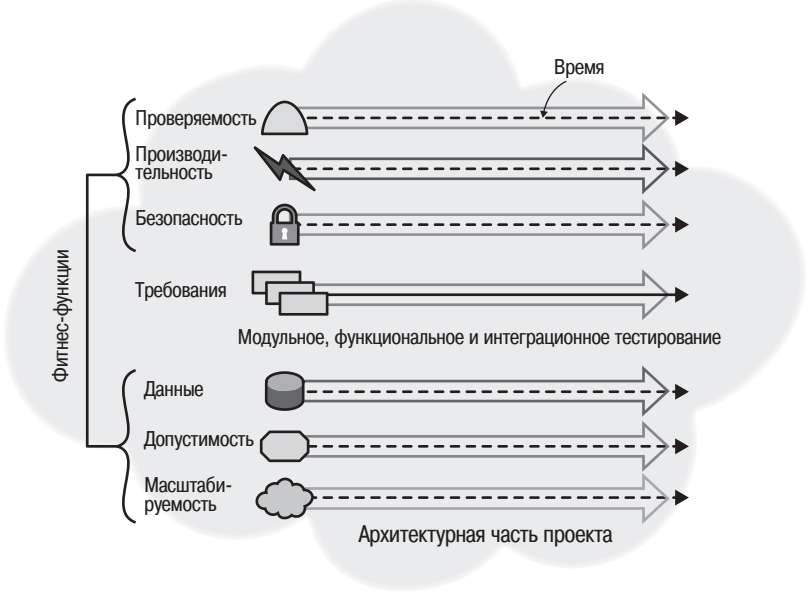


Рис. 1.2. Архитектура включает требования и другие измерения, каждое из которых защищено фитнес-функциями

На рис. 1.2 *проверяемость*, *данные*, *безопасность*, *производительность*, *допустимость* и *масштабируемость* выделены как характеристики архитектуры, важные для разрабатываемого приложения. Поскольку бизнес-требования со временем меняются, каждая из характеристик использует фитнес-функции для защиты своей целостности.

Хотя мы подчеркиваем важность целостного взгляда на архитектуру, мы также понимаем, что значительная часть эволюционирующей архитектуры относится к техническим паттернам и примыкающим к ним темам, таким как связанность, или сцепление (coupling), и связность (cohesion)[1]. В главе 5 мы обсудим, как связанность технической архитектуры влияет на ее способность эволюционировать, а в главе 6 рассмотрим влияние связанности данных.

Связанность относится не только к структурным элементам программных проектов. Многие компании-разработчики недавно обнаружили, что на архитектуру продукта удивительным образом влияет структура команды. Обычно мы говорим о связанности применительно к разрабатываемым продуктам, но влияние команды ощущается уже на ранних этапах разработки и весьма часто, поэтому мы обсудим и эту тему.

Эволюция архитектуры помогает ответить на два вопроса, часто возникающих у архитекторов, работающих в современной экосистеме: *Как осуществлять долгосрочное планирование, если все постоянно меняется?* и *Как, построив архитектуру, предотвратить постепенное ухудшение ее качества?* Рассмотрим эти вопросы более подробно.

Как осуществлять долгосрочное планирование, если все постоянно меняется?

Платформы разработки, которые мы используем, — пример постоянной эволюции. С появлением новых версий языков программирования совершенствуются API, повышается их гибкость и применимость; новые языки предлагают новую парадигму и новые наборы конструкций. Например, Java позиционировался как замена C++, облегчающая написание кода и управление памятью. Если взглянуть на то, что происходило последние 20 лет, можно заметить, что многие языки по-прежнему постоянно развивают свои API, в то время как для решения новых проблем появляются совершенно новые языки. Эволюция языков программирования показана на рис. 1.3.

[1] Речь идет о связанности между отдельными модулями и функциональной связанности (связности) элементов модуля. — *Примеч. науч.ред.*

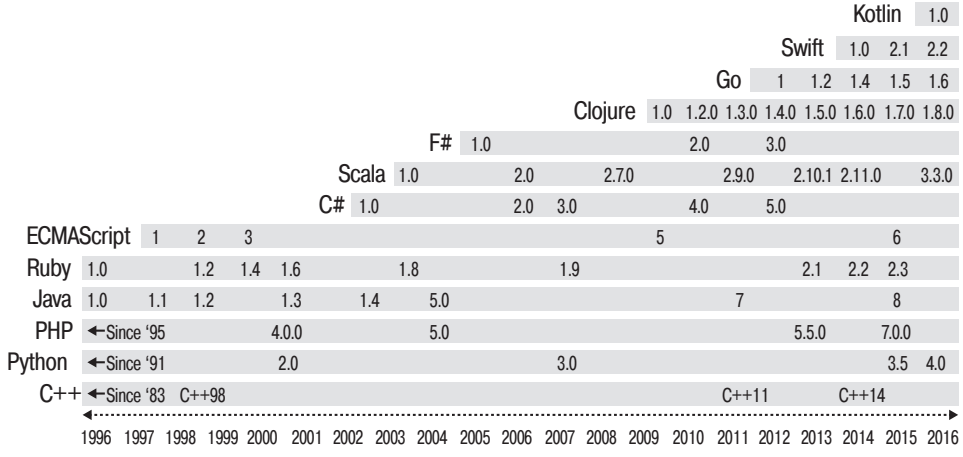


Рис. 1.3. Эволюция популярных языков программирования

Мы ожидаем постоянных изменений от всех аспектов разработки: платформ программирования, языков, операционной среды, технологий обеспечения персистентности, облачных решений и проч. Мы знаем, что изменения технологий или предметной области неизбежны, пусть и не можем предсказать, когда конкретно они произойдут или какие из них сохранятся. Следовательно, мы должны проектировать системы с учетом будущих изменений.

Если экосистема меняется неожиданно и изменения невозможно предсказать, какова *альтернатива* фиксированным планам? Архитекторы систем предприятий и разработчики должны адаптироваться. Традиционно стратегия долгосрочного планирования существовала не в последнюю очередь потому, что ПО стоило дорого. Однако современные подходы к разработке опровергают эту предпосылку, поскольку благодаря автоматизации ручных процессов и другим достижениям, таким как DevOps, изменения стали менее дорогостоящими.

За многие годы лучшие разработчики осознали, что одни части систем труднее изменять, чем другие. Вот почему *программная архитектура* определяется как «то, что трудно поддается дальнейшим изменениям». Это удобное определение делит изменения на те, которые осуществить легко, и те, которые осуществить действительно сложно. К сожалению, это определение также породило предвзятое представление об архитектуре: разработчики изначально уверены в том, что вносить изменения сложно, и это становится самореализующимся пророчеством.

Несколько лет назад архитекторы-новаторы вновь обратились к проблеме «трудных изменений»: что, если *встроить* способность изменения *в* архитек-

туру? Другими словами, если сделать *простоту изменений* основополагающим принципом архитектуры, то изменения перестанут быть трудными. Встраивание изменяемости в архитектуру, в свою очередь, ведет к возникновению совершенно нового набора моделей поведения, что снова нарушает динамическое равновесие.

Даже если экосистема не меняется, что делать с постепенной эрозией характеристик архитектуры? Архитекторы создают модели архитектуры, которые затем подвергаются воздействию реальных условий *реализации* того, что лежит поверх этих моделей. Как архитекторам защитить выделенные ими важные компоненты?

Как, построив архитектуру, предотвратить постепенное ухудшение ее качества

Во многих организациях происходят негативные процессы, часто называемые *«гниением»*. Архитекторы выбирают определенные архитектурные модели для обработки бизнес-требований и «-остей», но качество этих характеристик часто снижается со временем. Например, если архитектор создал многоуровневую архитектуру с верхним уровнем представления, нижним уровнем персистентных структур данных и несколькими промежуточными уровнями, то, чтобы повысить производительность, разработчики отчетов часто будут просить разрешения на прямой доступ к данным из уровня представления, минуя другие уровни. Архитекторы создают уровни, чтобы изолировать изменения. Затем разработчики обходят эти уровни, увеличивая связанность и делая выделение уровней бессмысленным.

Как архитекторам *защитить* выделенные ими характеристики архитектуры от разрушения? Добавление *способности эволюционировать (эволюционность)* в качестве характеристики подразумевает защиту других характеристик в ходе эволюции системы. Например, если архитектор разработал архитектуру для масштабируемости, качество этой характеристики не должно ухудшаться по мере развития системы. Таким образом, *способность эволюционировать* — это мета-характеристика, архитектурная обертка, которая защищает все остальные характеристики.

Механизм эволюционной архитектуры в значительной степени пересекается с проблемами и целями управления архитектурой — принципами, определяющими дизайн, качество, безопасность и другие вопросы. В этой книге описаны различные способы, позволяющие автоматизировать управление эволюционной архитектурой.

Почему архитектура эволюционная

Частый вопрос об эволюционной архитектуре касается самого ее определения: почему ее называют *эволюционной* архитектурой, а не как-то иначе? Другие возможные термины — *инкрементная*, *непрерывная*, *адаптивная*, *реактивная*, *эмерджентная* и другие. Но ни один из этих терминов не отражает всю полноту описываемого понятия. Определение эволюционной архитектуры, которое мы приводим здесь, включает две важнейшие характеристики: инкрементность и управляемость.

Термины *непрерывная* (continual), *адаптивная* (agile) и *эмерджентная* (emergent) отражают изменчивость во времени — несомненно, критически важную характеристику эволюционной архитектуры, но ни один из этих терминов не говорит о том, *как* изменяется архитектура или какой может быть желаемая конечная архитектура. Хотя все термины подразумевают изменяющуюся среду, ни один из них не описывает, как должна выглядеть архитектура. Слово *управляемая* в нашем определении указывает, какую архитектуру мы хотим получить, — нашу конечную цель.

Мы предпочитаем слово *эволюционная*, а не *адаптируемая*, потому что нас интересуют архитектуры, претерпевающие фундаментальные эволюционные изменения, а не те, которые были исправлены и адаптированы до возрастающей непонятной случайной сложности. *Адаптация* подразумевает поиск способа заставить что-то работать, неважно, насколько элегантно или долговечно найденное решение. Чтобы создавать архитектуры, которые действительно развиваются, архитекторы должны обеспечивать поддержку реальных и постоянных, а не временных изменений. Возвращаясь к метафоре из естествознания, *эволюция* — это процесс создания системы, которая соответствует своему назначению и способна выживать в постоянно меняющейся среде жизнедеятельности. Системы могут иметь отдельные адаптации, но архитекторы должны в первую очередь думать о способности к эволюции всей системы в целом.

Еще одно полезное для архитекторов сравнение — это сравнение *эволюционной архитектуры* и *эмерджентного дизайна*, и они должны осознавать, почему не существует понятия «эмерджентная архитектура». Одно из ошибочных представлений об agile-разработке — это то, что в ней якобы отсутствует архитектура: «Давайте просто начнем писать код, а архитектура подтянется по ходу дела». Однако это зависит от простоты проблемы. Рассмотрим строительство. Как мы уже говорили, если вы строите собачью будку, вам не нужна архитектура; вы можете пойти в хозяйственный магазин, купить доски и сколотить будку. Но если вы строите 50-этажное офисное здание, без архитектуры не обойтись! Точно так же, если вы создаете простую систему каталогов для небольшого количества пользователей, вы, скорее всего, обойдетесь без подробного планирования. Однако если вы проектируете высокопроизводительную систему для

большого числа пользователей, вам необходим план! Agile-архитектура — это не *отсутствие* архитектуры; это *отсутствие бесполезной* архитектуры — излишней бюрократии, которая не добавляет ценности процессу разработки.

Еще один фактор, усложняющий архитектуру ПО, — различие в проектируемой сложности. Часто речь идет не о *простой* и *сложной* системах, а о системах, которые сложны по-разному. Другими словами, каждая система имеет уникальный набор критериев успеха. Хотя мы обсуждаем архитектурные стили, такие как микросервисы, каждый стиль по своей сути — это основа для сложной системы, которая растет и становится непохожей ни на какую другую.

Аналогично, если архитектор строит очень простую систему, он может не сильно заботиться об архитектуре. Однако сложные системы требуют целенаправленного проектирования и наличия отправной точки. *Эмерджентность* предполагает, что вы можете начать с нуля, в то время как архитектура предоставляет строительные леса, или структуру, для остальных частей системы — то, с чего можно начать.

Концепция *эмерджентности* также подразумевает, что команды могут медленно продвигаться к идеальному архитектурному решению. Однако, как и в строительстве зданий, идеальной архитектуры не существует, есть только способы, которыми архитекторы приходят к различным компромиссам. Для успешной реализации большинства проблем подходит множество разных архитектурных стилей. Однако какие-то стили будут лучше соответствовать проблеме, подразумевая меньшее сопротивление и меньшее количество обходных путей.

Одна из ключевых особенностей эволюционной архитектуры состоит в том, что необходимость обеспечивать определенную структуру и управление для поддержки долгосрочных целей соединяется в ней с отсутствием формальностей и трения.

Итоги

Полезные программные системы не статичны. Они должны расти и меняться по мере изменения предметной области и развития экосистемы, предоставляя новые возможности и усложняясь. Архитекторы и разработчики могут обеспечивать плавную эволюцию программных систем, но они должны понимать, какие методы разработки для этого лучше использовать и как структурировать свою архитектуру, чтобы способствовать изменениям.

Кроме того, на архитекторов возложена задача управлять продуктами, которые они разрабатывают, и методами разработки, которые они используют. К счастью, механизмы, которые мы описываем и которые облегчают эволюцию, также позволяют автоматизировать действия по управлению ПО. В следующей главе мы подробно рассмотрим, как это сделать.

ГЛАВА 2

Фитнес-функции

Механика эволюционной архитектуры охватывает инструменты и методы, которые разработчики и архитекторы используют для создания систем, способных эволюционировать. Важным элементом этой механики является защитный механизм, называемый *фитнес-функцией,* архитектурный эквивалент модульного теста для части приложения, относящейся к предметной области. В этой главе дается определение фитнес-функции и объясняются категории и использование этого важного строительного блока.

> Эволюционная архитектура поддерживает *управляемые*, инкрементные изменения в разных измерениях.

Слово *«управляемый»* в определении указывает на существование цели, к которой архитектура должна стремиться или которую должна выражать. Мы заимствуем концепцию из области эволюционных вычислений, называемую фитнес-функцией. Фитнес-функция используется при разработке генетических алгоритмов для оценки того, как достигается заданная цель.

Эволюционные вычисления включают механизмы, благодаря которым решения формируются постепенно посредством мутаций — небольших изменений в каждом поколении программного продукта. В эволюционных вычислениях выделяют несколько типов мутаций. Один из них — *отбор методом рулетки* (roulette mutation): если в алгоритме используются константы, эта мутация будет выбирать новые числа, как на колесе рулетки в казино. Например, предположим, что разработчик проектирует генетический алгоритм для решения задачи о коммивояжере, чтобы найти кратчайший маршрут для посещения нескольких городов. Если разработчик заметит, что меньшие числа, полученные в результате отбора методом рулетки, дают лучшие результаты, он может построить фитнес-функцию, которая будет направлять «процесс принятия решения» во время мутации. Таким образом, фитнес-функция используется для оценки того, насколько решение близко к идеальному.

Что такое фитнес-функция

Определение фитнес-функции в архитектуре мы будем строить на основе концепции фитнес-функции в эволюционных вычислениях:

> Архитектурная фитнес-функция — это механизм, который обеспечивает объективную оценку целостности выбранной характеристики (характеристик) архитектуры.

Архитектурные фитнес-функции формируют первичные механизмы для реализации эволюционной архитектуры.

Развивая часть решения, относящуюся к *предметной области*, команды разрабатывали разнообразные инструменты и методы для управления интеграцией новых возможностей без нарушения работы существующих. Это модульное, функциональное и приемочное тестирование. В большинстве крупных компаний существует целый отдел, занимающийся управлением эволюцией предметной области, называемый отделом *обеспечения качества* (*QA*, quality assurance): он гарантирует, что изменения не окажут негативного влияния на существующую функциональность.

Таким образом, в грамотно организованных командах существуют механизмы управления эволюционными изменениями предметной области: добавление новых функций, изменение поведения и т. д. Предметная область, как правило, пишется на достаточно согласованном технологическом стеке: *Java*, *.NET* или других платформах. Следовательно, команды могут загружать и применять библиотеки тестирования, подходящие для используемого набора стеков.

Фитнес-функции относятся к характеристикам архитектуры так же, как модульные тесты относятся к предметной области. Но нельзя загрузить единственный инструмент, подходящий для всех проверок характеристик архитектуры. Скорее фитнес-функции вмещают широкий спектр инструментов в различных частях экосистемы, в зависимости от характеристик архитектуры, которыми управляет команда (рис. 2.1).

Как показано на рис. 2.1, архитекторам доступны различные инструменты для определения фитнес-функций.

Мониторы

Инструменты операций и DevOps, такие как мониторы, позволяют командам отслеживать производительность, масштабируемость и другие показатели.

Метрики кода

Архитекторы могут встраивать проверки метрик и прочие проверки в модульные тесты, затрагивающие широкий спектр особенностей архитектуры, включая критерии проектирования (примеры см. в главе 4).

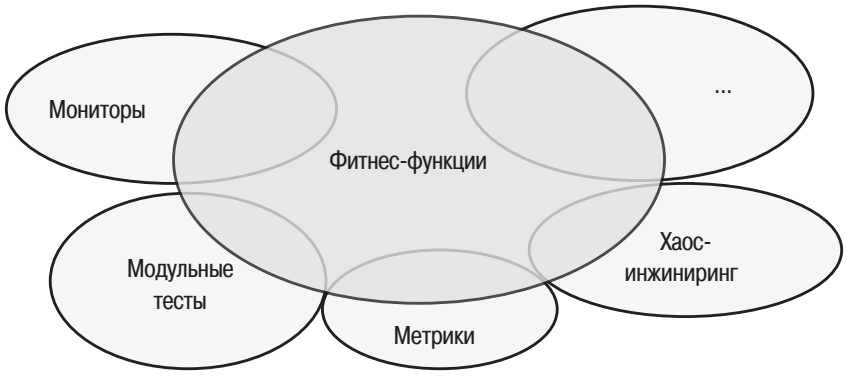


Рис. 2.1. Фитнес-функции охватывают широкий спектр инструментов и методов

Хаос-инжиниринг (chaos engineering)

В рамках этого нового инженерного направления команды искусственно вызывают сбои в удаленных средах, чтобы повысить отказоустойчивость разрабатываемых систем.

Фреймворки тестирования архитектуры

В последние годы возникают все новые фреймворки тестирования структурной области архитектуры, позволяющие написать множество проверок в автоматизированных тестах.

Сканирование безопасности

Безопасность, даже если за нее отвечает другая часть организации, влияет на проектные решения, которые принимают архитекторы, и, таким образом, попадает в перечень проблем, которыми архитекторам необходимо управлять.

Прежде чем определять категории фитнес-функций и другие понятия, рассмотрим пример, который поможет представить их концепцию в менее абстрактном виде. *Цикл компонентов* — это общий антипаттерн для всех платформ и их составных частей. Рассмотрим три компонента на рис. 2.2.

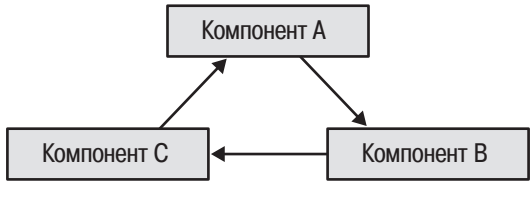


Рис. 2.2. Цикл возникает, когда компоненты находятся в циклической зависимости

Архитекторы считают циклическую зависимость, показанную на рис. 2.2, антипаттерном, поскольку она затрудняет повторное использование какого-либо из компонентов — оставшиеся компоненты зависимости также должны быть использованы. Таким образом, в целом архитекторы стремятся минимизировать количество циклов. Однако Вселенная активно препятствует этому и использует в своих интересах инструменты, повышающие удобство работы. Что происходит в современной IDE, когда разработчик ссылается на класс, на пространство имен/пакет которого он еще не ссылался? Появляется диалоговое окно с предложением автоматического импорта необходимого пакета.

Разработчики настолько привыкли к этой возможности, что рефлекторно принимают предложение. В большинстве случаев автоимпорт очень удобен и не вызывает проблем. Однако иногда он формирует цикл компонентов. Как архитекторы борются с этим?

Рассмотрим набор пакетов, показанный на рис. 2.3.

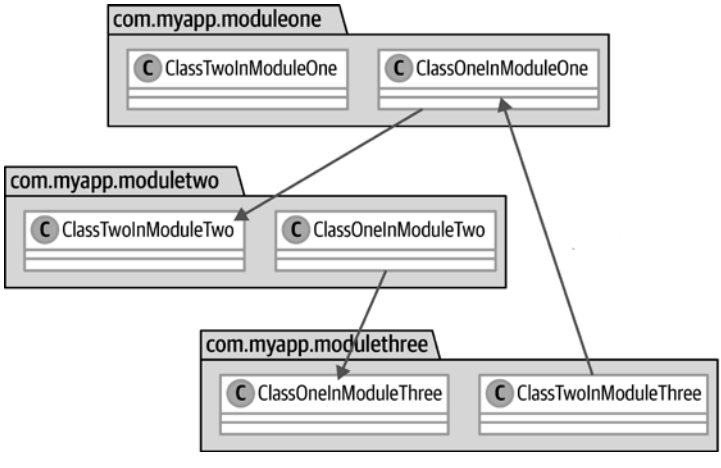


Рис. 2.3. Циклы компонентов, представленных в виде пакетов Java

ArchUnit (https://www.archunit.org/) — это инструмент тестирования, созданный на основе JUnit (и использующий некоторые его возможности), применяемый для тестирования различных функций архитектуры, включая проверки на наличие циклов в определенной области видимости, как показано на рис. 2.3.

В примере 2.1 показано, как с помощью ArchUnit предотвратить формирование циклов.

Пример 2.1. Предотвращение циклов с помощью ArchUnit

```
public class CycleTest {
    @Test
    public void test_for_cycles() {
        slices().
          matching("com.myapp.(*)...").
          should().beFreeOfCycles();
}
```

В этом примере инструмент тестирования распознает циклы. Архитектор, желающий предотвратить появление циклов в своей кодовой базе, может подключить это тестирование к процессу непрерывной сборки и навсегда забыть о циклах. Больше примеров использования ArchUnit и похожих инструментов см. в главе 4.

Для начала дадим более строгое определение фитнес-функций, а затем рассмотрим, как они управляют общей эволюцией архитектуры.

Отметим, что слово *функция* в определении не обязывает архитекторов выражать все фитнес-функции в коде. Говоря математическим языком, функция принимает данные из набора допустимых входных значений и выдает результат из набора допустимых выходных значений. В разработке термин *«функция»* также обычно используется для обозначения чего-то, что реализуется в коде. Однако, как и в случае с критериями приемки в agile-разработке, фитнес-функции для эволюционной архитектуры могут быть не реализуемы в продукте (например, из-за необходимости обеспечить нормативное соответствие требуется ручной процесс). Архитектурная фитнес-функция — это *объективная мера*, которую можно реализовать разными способами.

Как говорилось в главе 1, архитектура реального мира включает различные измерения, в том числе требования к производительности, надежности, безопасности, работоспособности, стандартам кодирования и интеграции. Фитнес-функция необходима для каждого из этих требований, и это вынуждает нас искать (а иногда и создавать) способы измерения параметров, которыми мы хотим управлять. Мы рассмотрим несколько примеров, а затем обратимся к различным видам функций.

Фитнес-функции отлично подходят для требований производительности. Рассмотрим требование, согласно которому время ответа по всем запросам на обслуживание не должно превышать 100 мс. Можно реализовать тест (то есть фитнес-функцию), который измеряет время ответа по запросу на обслуживание и завершается ошибкой, если результат превышает 100 мс. Для этого в набор тестов для каждого нового сервиса необходимо добавить соответствующий тест производительности (подробнее о запуске фитнес-функций вы узнаете

в главе 3). На примере производительности также удобно показать, что в рамках одного и того же требования можно измерять разные параметры. Например, *производительность* может означать время запроса/ответа, измеряемое инструментом мониторинга, или такую метрику, как *first contentful paint* (первая отрисовка контента) в Lighthouse (https://developer.chrome.com/docs/lighthouse/) — метрику производительности мобильных устройств. Цель фитнес-функции производительности — измерить не *все* параметры производительности, а только те, которыми архитекторы считают важным управлять.

Фитнес-функции также можно использовать для соблюдения стандартов кодирования. Общей метрикой кода является цикломатическая сложность — мера сложности функций или методов во всех структурированных языках программирования. Используя один из инструментов оценки этой метрики, архитекторы могут установить порог верхнего значения, соблюдение которого будет проверять модульный тест, выполняемый в процессе непрерывной интеграции.

Сложность или другие ограничения иногда не дают разработчикам полностью реализовать фитнес-функции. Рассмотрим, например, восстановление базы данных после серьезного сбоя. При том что само восстановление может (и должно) быть полностью автоматизировано, запуск самого теста, скорее всего, лучше проводить вручную. Кроме того, гораздо эффективнее вручную определять, успешно ли пройден тест, хотя разработчикам все же не следует забывать о скриптах и автоматизации.

Эти примеры наглядно иллюстрируют, что фитнес-функции могут принимать совершенно разные формы и немедленно реагировать на отказ, и даже показывают, когда и как разработчики могут их запускать. Хотя нельзя запустить один-единственный сценарий и сказать: «Текущая общая оценка пригодности архитектуры — 42». Мы можем получить вполне определенное и однозначное понятие о состоянии архитектуры. И кроме того, можем обсуждать возможные изменения этого состояния.

Наконец, когда мы говорим, что эволюционная архитектура управляется фитнес-функцией, мы имеем в виду следующее. Мы оцениваем отдельные архитектурные решения по фитнес-функциям, которые применимы только к этому решению или ко всей системе, чтобы определить влияние изменений. Фитнес-функции в совокупности определяют, что для нас важно в архитектуре, позволяя принимать важные и сложные компромиссные решения, касающиеся разработки программных систем.

Вы можете подумать: «Но мы уже много лет оцениваем метрики кода в рамках непрерывной интеграции — что тут нового?!» И будете правы: идея проверки частей ПО в рамках автоматизированного процесса так же стара, как и сама автоматизация. Однако раньше мы рассматривали механизмы проверки архи-

тектуры по отдельности: качество кода, метрики DevOps, безопасность и так далее. Фитнес-функции объединяют многие существующие понятия в единый механизм, позволяя архитекторам представлять многие применяемые (часто ситуативные) тесты «нефункциональных требований» в единой манере. Представление важных пороговых значений и требований к архитектуре в виде фитнес-функций позволяет четче сформулировать ранее расплывчатые, субъективные критерии оценки. Для создания фитнес-функций мы используем множество инструментов, включая традиционные средства тестирования, мониторинг и другие. Не все тесты представляют собой фитнес-функции, но некоторые таковыми являются — если тест оценивает целостность архитектурных элементов, мы считаем его фитнес-функцией.

Категории

Фитнес-функции можно классифицировать по их масштабу, периодичности, результату, способу вызова, проактивности и охвату.

Масштаб: атомарные и комплексные функции

Атомарные (atomic) фитнес-функции используются в единичном контексте и затрагивают одно конкретное свойство архитектуры. Отличный пример атомарной фитнес-функции — модульный тест, который проверяет выбранную архитектурную характеристику, например связанность модулей (пример фитнес-функции такого типа мы покажем в главе 4). Таким образом, некоторые тесты уровня приложения можно считать фитнес-функциями, но не все модульные тесты — это фитнес-функции, а только те, которые проверяют характеристики архитектуры. На рис. 2.3 представлена *атомарная* фитнес-функция: она проверяет только наличие циклов между компонентами.

Для некоторых характеристик архитектуры необходимо тестировать не только их отдельные измерения. *Комплексные* (holistic) фитнес-функции используются в общем контексте и проверяют совокупность свойств архитектуры. Они необходимы для проверки того, что комбинации атомарных функций останутся рабочими в реальных условиях. Представьте, например, что в архитектуре имеются фитнес-функции безопасности и масштабируемости. Для фитнес-функции безопасности важна релевантность данных, а для фитнес-функции масштабируемости — количество пользователей, одновременно работающих в системе, в определенном диапазоне времени задержки. Для достижения масштабируемости разработчики внедряют кэширование, позволяющее атомарной фитнес-функции масштабируемости успешно пройти тест. Когда кэширование не включено, функция безопасности проходит тест. Однако при совместном

запуске кэшированные данные считаются устаревшими для функции безопасности, и комплексный тест не проходит.

Очевидно, что протестировать все возможные комбинации элементов архитектуры — невыполнимая задача, поэтому архитекторы выборочно используют комплексные фитнес-функции для тестирования важных взаимодействий. Такая избирательность и расстановка приоритетов также позволяют архитекторам и разработчикам оценить сложность реализации конкретного сценария тестирования и, следовательно, ценность рассматриваемой характеристики. Зачастую качество архитектуры определяется взаимодействиями между ее элементами, являющимися предметом комплексных фитнес-функций.

Периодичность: триггерные, непрерывные и временные функции

Периодичность выполнения — еще одна характеристика фитнес-функции.

Триггерные (triggered) фитнес-функции запускаются на основе определенного события. Например, когда разработчик выполняет модульный тест, пайплайн развертывания запускает модульные тесты или сотрудник отдела контроля качества (QA) проводит исследовательское тестирование. Сюда входит и традиционное тестирование: модульные, функциональные тесты и тесты на основе поведения (BDD, behavior-driven development).

Непрерывные (continual) тесты проводятся не по графику, а подразумевают постоянную проверку архитектурного свойства (свойств), например скорости транзакций. Рассмотрим, к примеру, микросервисную архитектуру, в которой необходимо построить фитнес-функцию времени транзакции — сколько времени в среднем требуется для завершения транзакции. Тесты, запускаемые в пайплайне (триггерные тесты), мало скажут о реальном поведении. Поэтому архитекторы, обычно используя технику, называемую синтетическими транзакциями (synthetic transactions), строят непрерывную фитнес-функцию, которая имитирует транзакцию на продакшене — в условиях, когда выполняются остальные реальные транзакции. Это позволяет проверить поведение и собрать реальные данные о системе, работающей «в естественных условиях».

Обратите внимание, что, если вы используете инструмент мониторинга, это не значит, что у вас появляется фитнес-функция, которая должна давать *объективные результаты*. Скорее добавление *оповещения для отклонений от объективной величины метрики* в инструмент мониторинга превращает его в фитнес-функцию.

Разработка на основе мониторинга (MDD, monitoring-driven development) — еще одна техника тестирования, набирающая популярность. Для проверки

результатов работы системы MDD не полагается исключительно на тесты, а использует мониторы на продакшене для оценки состояния технологической части продукта и части, относящейся к бизнесу. Эти непрерывные фитнес-функции более динамичны, чем стандартные триггерные тесты, и относятся к широкой категории *архитектуры, управляемой фитнес-функциями,* которая более подробно рассматривается в главе 7.

СИНТЕТИЧЕСКИЕ ТРАНЗАКЦИИ

Как команды измеряют сложные, реальные взаимодействия между сервисами в микросервисной архитектуре? Один из распространенных методов — *синтетические транзакции*. В рамках этого метода запросы в систему помечаются флагом, который указывает, что выбранная транзакция является синтетической. Далее обычный ход взаимодействий в архитектуре (обычно отслеживаемых с помощью идентификатора корреляции для ретроспективного анализа) в точности повторяется до последнего шага, на котором система видит флаг и не фиксирует транзакцию как реальную. Это позволяет архитекторам и DevOps точно узнать, как работает их сложная система.

Ни один рассказ о синтетических транзакциях не будет полным без упоминания о сотнях программных средств, появившихся случайно, потому что кто-то забыл установить флаг «синтетический», который сам может управляться фитнес-функцией. Проверьте, что все фитнес-функции, идентифицированные как синтетические транзакции (например, с помощью аннотации), имеют этот флаг!

Хотя большинство фитнес-функций работают либо при изменениях, либо постоянно, в некоторых случаях архитекторы могут добавлять в оценку пригодности компонент времени, что приводит к появлению *временных* (temporal) фитнес-функций. Например, если в проекте используется библиотека шифрования, архитектор может создать временную фитнес-функцию для напоминания о необходимости проверки выполнения важных обновлений. Эти фитнес-функции также часто применяются в тестах на *ошибки при обновлении (break upon upgrade)*. Работая на таких платформах, как Ruby on Rails, некоторые разработчики с нетерпением ждут новых релизов и вместе с ними новых функций, поэтому добавляют функцию в текущую версию через *бэкпорт* (back port), пользовательскую реализацию будущей функции. А когда версия проекта наконец обновляется, возникают проблемы, потому что бэкпорт часто оказывается несовместим с «реальной» версией. Разработчики используют тесты *break upon upgrade* для обертывания бэкпорт-функций, чтобы перепроверить их после обновления.

Временные фитнес-функции также часто используются в связи с одной важной, но отложенной потребностью, которая рано или поздно возникает практически в каждом проекте. Многие разработчики сталкивались с необходимостью обновлять сразу несколько версий базового фреймворка или библиотеки, от которых зависит их проект, — между крупными релизами происходит так много изменений, что часто довольно сложно перескочить от одного к другому. Однако обновление основного фреймворка требует много времени и не считается критичным, поэтому их часто пропускают, и фреймворк устаревает. Архитекторы могут использовать временную фитнес-функцию в сочетании с такими инструментами, как Dependabot (https://github.com/dependabot) или snyk (https://snyk.io/), которые отслеживают релизы, версии и исправления безопасности, и генерить все более настойчивые напоминания о необходимости провести обновление после того, как будут выполнены корпоративные задачи (например, выпущен релиз первого патча).

Практический пример: триггерная или непрерывная?

Часто выбор между *непрерывной* и *триггерной* фитнес-функциями сводится к компромиссу между подходами. Многие разработчики распределенных систем типа микросервисов стремятся добиться такой же проверки зависимостей, но на разрешенных взаимодействиях между сервисами, а не на циклах. Рассмотрим набор сервисов, показанный на рис. 2.4: это более продвинутая версия фитнес-функции циклической зависимости, изображенной на рис. 2.3.

На рис. 2.4 архитектор спроектировал систему так, что в оркестраторе заключено состояние рабочего процесса. Если какие-либо из сервисов будут взаимодействовать друг с другом в обход оркестратора, команда не будет иметь точной информации о состоянии рабочего процесса.

В случае циклов зависимостей существуют инструменты, позволяющие архитекторам проводить проверки во время компиляции. Однако сервисы не ограничены одной платформой или технологическим стеком, поэтому крайне маловероятно, что кто-то уже создал инструмент, который точно соответствует конкретной архитектуре. Это пример того, о чем мы уже говорили: часто лучше создавать свои собственные инструменты, а не полагаться на сторонние. Для конкретной системы архитектор может построить либо *непрерывную*, либо *триггерную* фитнес-функцию.

При построении непрерывной функции архитектор должен убедиться, что каждый из сервисов предоставляет информацию для мониторинга (обычно через определенный порт) о том, к какому компоненту сервис обращается в ходе работы. Оркестратор либо вспомогательный сервис отслеживает эти сообщения,

чтобы исключить неразрешенное взаимодействие. В качестве альтернативы вместо мониторов команда может использовать асинхронные очереди сообщений, когда сервисы каждой предметной области публикуют в очереди сообщение с информацией о взаимодействии, а оркестратор прослушивает эту очередь и проверяет взаимодействия. Такая фитнес-функция является непрерывной, поскольку принимающий сервис должен немедленно реагировать на неразрешенное взаимодействие. Например, эта ошибка может указывать на проблему безопасности или другой негативный побочный эффект.

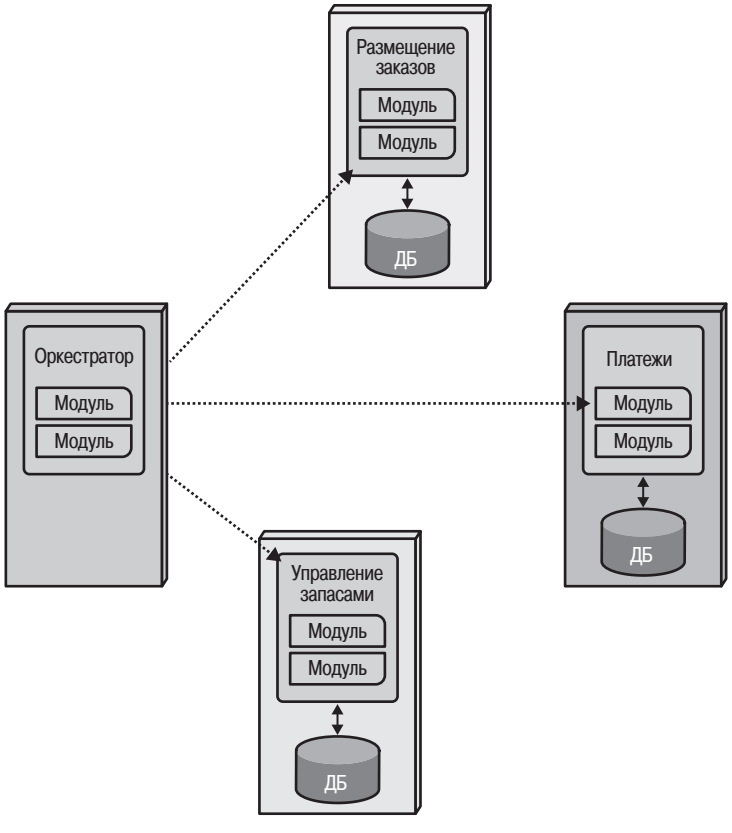


Рис. 2.4. Набор оркестрированных микросервисов, в котором сервисы, не являющиеся частью оркестратора, не должны взаимодействовать между собой

Немедленное реагирование является преимуществом фитнес-функций этого типа: архитекторы и другие заинтересованные стороны сразу же узнают о нарушении. Однако такое решение добавляет накладные расходы выполнения: мониторы и/или очереди сообщений требуют операционных ресурсов, и такой

уровень отслеживаемости может негативно сказаться на производительности, масштабируемости и других показателях.

В качестве альтернативы команда может реализовать триггерную версию фитнес-функции. В этом случае пайплайн развертывания периодически вызывает фитнес-функцию, которая собирает файлы журналов и исследует взаимодействия, определяя, все ли они соответствуют требованиям. Мы показываем реализацию этой фитнес-функции в разделе «Управление взаимодействием в микросервисах» на с. 94 (глава 4). Преимущество в этом случае заключается в том, что время выполнения не будет иметь никакого влияния — функция запускается только триггером и просматривает записи журнала. Однако не следует использовать триггерный тип для критически важных сфер, таких как безопасность, где задержка во времени может быть недопустимой.

Как и во всем, что касается программной архитектуры, выбор между триггерными и непрерывными фитнес-функциями часто определяет различные компромиссы, что делает это решение индивидуальным для каждого конкретного случая.

Результат: статические и динамические функции

Статические фитнес-функции предполагают фиксированный результат, например двоичный результат модульного теста — *пройден/не пройден*. К ним относятся все фитнес-функции с предварительно заданным требуемым значением: двоичным, диапазоном чисел, включением множества и так далее. В качестве фитнес-функций часто используются метрики. Например, архитектор может задать приемлемые диапазоны средней цикломатической сложности методов в кодовой базе.

В основе *динамических* фитнес-функций лежит сдвиг оценки при появлении дополнительного контекста, часто в реальном времени. Например, рассмотрим фитнес-функцию для проверки масштабируемости и отзывчивости на запросы/ответы для заданного количества пользователей. При увеличении числа одновременно работающих пользователей допускается небольшое снижение скорости отклика, но не настолько, чтобы это вызвало неудобства. Таким образом, фитнес-функция отзывчивости будет учитывать количество одновременно работающих в системе пользователей и соответствующим образом корректировать оценку.

Обратите внимание, что свойства *динамический* и *объективный* не противоречат друг другу — фитнес-функции должны оценивать объективный результат, но эта оценка может быть основана на динамической информации.

Вызов: автоматический или ручной

Архитекторы любят автоматизацию — она входит в инкрементные изменения, и мы подробно рассмотрим этот вопрос в главе 3. Поэтому неудивительно, что большинство фитнес-функций будут выполняться в автоматизированном контексте: непрерывной интеграции, пайплайнах развертывания и так далее. Действительно, разработчики и инженеры DevOps проделали огромную работу в рамках непрерывной доставки, чтобы автоматизировать многие части экосистемы разработки, для которых автоматизация считалась невозможной.

Однако, как бы нам ни хотелось автоматизировать вообще всю разработку, некоторые ее части оказывают сопротивление. Иногда невозможно автоматизировать критически важный параметр системы, например требования законодательства или исследовательское тестирование, что приводит к необходимости выполнять фитнес-функции вручную. Другая аналогичная ситуация — проект имеет эволюционный потенциал, но соответствующие инженерные практики еще отсутствуют. Например, возможно, в конкретном проекте большинство тестов все еще выполняется вручную и продолжит так выполняться в ближайшем будущем. В обоих этих случаях (и в других) используются *ручные* фитнес-функции, которые проверяются с участием человека.

Повышение эффективности подразумевает максимальное исключение ручных операций, но во многих проектах по-прежнему без них не обойтись. Поэтому мы продолжаем определять фитнес-функции для этих характеристик и проверять их вручную в пайплайнах развертывания (более подробно об этом говорится в главе 3).

Проактивность: преднамеренные и эмерджентные функции

Хотя архитекторы задают большинство фитнес-функций еще на начальном этапе проекта, когда определяют характеристики архитектуры, некоторые фитнес-функции добавляются в процессе разработки системы. Все важные части архитектуры никогда не бывают известны с самого начала (классическая проблема *неизвестных неизвестных,* которую мы рассмотрим в главе 7), поэтому приходится определять фитнес-функции по мере развития системы. Архитекторы пишут *преднамеренные* (intentional), продуманные фитнес-функции в начале работы над проектом, в рамках формального управления, иногда совместно с другими архитекторами, например архитекторами системы предприятия.

Фитнес-функции не только проверяют первоначальные предположения архитекторов, но и обеспечивают постоянное управление. Это позволяет замечать поведение, которое можно улучшить, — и для этого создаются *эмерждентные* фитнес-функции. Необходимо внимательно отслеживать отклонения поведения в проекте, особенно те, которые можно проверить с помощью фитнес-функций, и оперативно добавлять необходимые фитнес-функции.

Функции двух этих видов иногда образуют совокупность, изначально создававшуюся в целях защиты какого-то параметра, но со временем превратившуюся в частично или даже полностью иную фитнес-функцию. Как и модульные тесты, фитнес-функции становятся частью кодовой базы. Таким образом, по мере изменения и развития архитектурных требований соответствующие фитнес-функции также должны изменяться.

Охват: нужны ли фитнес-функции, специфичные для конкретной предметной области

Нас иногда спрашивают, существуют ли зависимости между предметными областями и определенными архитектурными фитнес-функциями. Хотя в архитектуре ПО нет ничего невозможного и один и тот же фреймворк автоматизированного тестирования можно использовать для реализации нескольких фитнес-функций, обычно фитнес-функции используются только для абстрактных принципов архитектуры, а не для предметной области (бизнес-логики). На практике же, если применяются одни и те же средства автоматизации тестирования, мы наблюдаем разделение тестов. Один набор предназначается для тестирования логики предметной области (например, традиционные модульные или сквозные тесты), а другой — для тестирования фитнес-функций (например, тесты производительности или масштабируемости).

Такое практическое разделение позволяет избежать дублирования и лишних усилий. Помните, что фитнес-функции — это еще один механизм проверки в рамках проекта, который должен быть совместим с другими проверками (предметной области). Чтобы избежать дублирования работы, командам стоит ограничить фитнес-функции областью архитектуры, а для проверки бизнес-логики выделить другие тесты. Рассмотрим, к примеру, *эластичность*, которая описывает способность сайта справляться с внезапным наплывом пользователей. Заметьте, что параметр эластичности лежит в плоскости архитектуры, а сайт может быть любым — онлайн-игра, онлайн-магазин или онлайн-кинотеатр. Таким образом, эта часть архитектуры управляется фитнес-функцией. Напротив, ситуации наподобие смены адреса требуют знания предметной области и традиционных механизмов тестирования. Такой метод помогает определить ответственность за тесты.

Итак, набор необходимых фитнес-функций трудно предсказать даже в распространенных предметных областях (таких как финансы). Для каждой команды и проекта в конечном итоге важными и ценными могут быть абсолютно разные параметры.

Кто пишет фитнес-функции

Фитнес-функции представляют собой архитектурный аналог модульных тестов и с точки зрения разработки и инженерной практики должны рассматриваться таким же образом. Как правило, архитекторы пишут фитнес-функции по мере того, как определяют объективные значения важных характеристик архитектуры. И архитекторы, и разработчики поддерживают фитнес-функции, в том числе следя за тем, чтобы они успешно проходили тесты — это является объективной мерой пригодности архитектуры.

Архитекторы *должны* совместно с разработчиками определять и изучать назначение и применимость фитнес-функций, что добавляет дополнительный уровень проверки к качеству системы в целом. Поэтому иногда, если изменения нарушат правила управления, фитнес-функции будут проваливать тесты — и это хорошо! Однако разработчики должны понимать назначение фитнес-функции, чтобы устранить неисправность и продолжить сборку. Для архитекторов и разработчиков очень важно работать совместно, чтобы разработчики рассматривали управление не как дополнительное бремя, а как полезное ограничение для защиты важной функциональности.

Распространяйте информацию о ключевых и значимых фитнес-функциях, публикуя результаты выполнения фитнес-функций где-нибудь в доступном всем месте или в общем пространстве, чтобы разработчики не забывали о них, когда пишут код.

Фреймворк тестирования фитнес-функций

Для тестирования бизнес-логики существуют разнообразные *платформенно-зависимые* инструменты, поскольку функциональный модуль пишется на определенной платформе/технологическом стеке. Например, если основной язык — Java, разработчики могут выбирать из широкого набора инструментов и фреймворков для модульного, функционального, пользовательского и других видов тестирования. Соответственно, архитекторы ищут поддержки такого же уровня, «под ключ», и для архитектурных фитнес-функций — а она, как правило,

отсутствует. В главе 4 мы рассмотрим несколько простых инструментов для загрузки и запуска фитнес-функций, но таких инструментов мало по сравнению с библиотеками тестирования предметной области. В основном это связано с тем, что фитнес-функции очень разнообразны, как показано на рис. 2.1: для эксплуатационных фитнес-функций требуются инструменты мониторинга, для фитнес-функций безопасности — инструменты сканирования, для проверки качества — метрики на уровне кода и так далее. Во многих случаях не существует подходящего инструмента для конкретной архитектурной конфигурации. Как мы покажем в последующих главах, архитекторы могут использовать программный «клей», чтобы относительно легко составлять полезные фитнес-функции, но все же усилий придется приложить больше, чем при загрузке готового фреймворка.

Результаты и реализация

Архитекторам важны результаты — объективные величины характеристик архитектуры, а не детали реализации. Архитекторы часто пишут фитнес-функции в технологических стеках, отличающихся от основной платформы предметной области, либо используют инструменты DevOps или любой другой процесс, удобный для объективного измерения интересующего предмета. Важная аналогия с *функцией* в термине *фитнес-функция* подразумевает нечто, что принимает входные данные и производит выходные без побочных эффектов. Подобным же образом фитнес-функция измеряет *результат* — объективную оценку заданных характеристик архитектуры.

В книге мы приводим примеры реализации фитнес-функций, но для читателей важно сосредоточиться на результате и на том, *почему* мы что-то измеряем, а не на том, *как* производить то или иное измерение.

Хотя архитекторов интересует изучение эволюционных архитектур, мы не стремимся моделировать биологическую эволюцию. Теоретически мы могли бы построить архитектуру, которая случайным образом изменяет один из своих битов (мутация) и заново себя развертывает. Через несколько миллионов лет мы, вероятно, получили бы очень интересную архитектуру. Однако мы не можем ждать миллионы лет.

Нам необходимо, чтобы наша архитектура развивалась направленно, поэтому мы накладываем ограничения на различные ее характеристики, чтобы сдержать нежелательные ответвления эволюции. Хороший пример — разведение собак: выбирая нужные характеристики, за относительно короткое время можно добиться появления у собак клыков совершенно разной формы.

PENULTIMATEWIDGETS И ЭЛЕКТРОННАЯ ТАБЛИЦА ХАРАКТЕРИСТИК АРХИТЕКТУРЫ СИСТЕМЫ ПРЕДПРИЯТИЯ

Когда архитекторы PenultimateWidgets решили построить новую проектную платформу, они сначала создали электронную таблицу, где указали все требуемые характеристики: масштабируемость, безопасность, отказоустойчивость и множество других понятий, оканчивающихся на «-ость». Но затем они столкнулись с вечным вопросом: как гарантировать, что новая архитектура действительно будет поддерживать эти характеристики? Как предотвратить снижение качества этих важных характеристик по мере добавления новых функций?

Решением стало создание фитнес-функций для каждой характеристики из электронной таблицы, для чего некоторые характеристики пришлось переформулировать для соответствия объективным критериям оценки. Вместо периодической спонтанной проверки на соответствие важным критериям разработчики подключили фитнес-функции к пайплайну развертывания (более подробно об этом рассказывается в главе 3).

Можно также рассматривать *общесистемную фитнес-функцию* как набор фитнес-функций, каждая из которых соответствует одному или нескольким измерениям архитектуры. Использование общесистемной фитнес-функции помогает понять необходимые компромиссы, когда отдельные элементы функции вступают в противоречие друг с другом. Как это часто бывает при решении многофункциональных задач оптимизации, существует риск столкнуться с невозможностью оптимизировать все значения одновременно, что вынуждает выбирать. Например, применительно к архитектурным фитнес-функциям такая характеристика, как производительность, может вступать в конфликт с безопасностью из-за стоимости шифрования. Это классический пример проклятия всех архитекторов — *компромисса*. Компромиссы — основная причина головной боли архитектора, пытающегося найти баланс противоположных сил, в данном случае масштабируемости и производительности. Архитекторам приходится иметь дело с вечной проблемой сравнения этих принципиально отличающихся друг от друга характеристик (сравнение яблока с апельсином), а все заинтересованные стороны считают, что именно их интересы важнее всего. Общесистемные фитнес-функции позволяют использовать единый объединяющий механизм фитнес-функций для разных параметров, фиксируя и защищая важные характеристики архитектуры. Взаимосвязь между общесистемной фитнес-функцией и входящими в нее более мелкими фитнес-функциями показана на рис. 2.5.

Общесистемная фитнес-функция имеет решающее значение для создания архитектуры, способной к эволюции, поскольку нам нужна основа, позволяющая

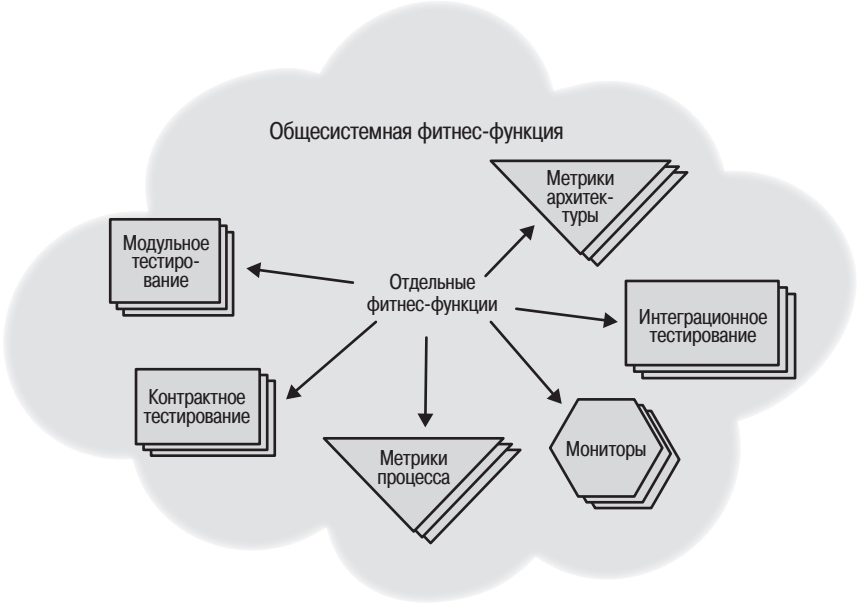


Рис. 2.5. Общесистемные и отдельные фитнес-функции

сравнивать и оценивать архитектурные характеристики. В отличие от более направленных фитнес-функций архитекторы едва ли будут пытаться «оценить» общесистемную фитнес-функцию. Скорее она служит для определения приоритетов в будущей архитектуре. Хотя фитнес-функции могут и не указать на лучший возможный компромисс, но благодаря оценке объективных параметров они помогают архитекторам лучше понять, какие силы имеют наибольшее влияние в системе, и определить, какие компромиссы необходимы для системы в целом.

> Система никогда не является суммой своих частей. Она является продуктом их взаимодействия.
>
> — *Д-р Рассел Акофф (Russel Ackoff)*

В отсутствие управления эволюционная архитектура становится просто реакционной архитектурой. Таким образом, приступая к разработке архитектурного решения для любой системы, необходимо в первую очередь определить ее важные измерения, такие как масштабируемость, производительность, безопасность, схемы данных и т. д. Концептуально это позволит архитекторам взвесить значимость фитнес-функции исходя из ее важности для общего поведения системы.

Итоги

Идея применения фитнес-функций в программной архитектуре пришла к Ребекке, когда она поняла, что может использовать некоторые из своих наработок, полученных в другой технической области (эволюционных вычислениях), и применить их для создания ПО. Архитекторы всегда тестировали отдельные части архитектуры, но раньше они не объединяли все применяемые методы тестирования в единую всеобъемлющую концепцию. Рассмотрение всех различных инструментов и методов управления как фитнес-функций позволяет командам сконцентрироваться на выполнении задач.

В следующей главе мы рассмотрим другие особенности реализации фитнес-функций.

ГЛАВА 3

Инкрементные изменения архитектуры

В 2010 году Джез Хамбл (Jez Humble) и Дэйв Фарли (Dave Farley) выпустили книгу «Continuous Delivery»[1] (http://continuousdelivery.com/) — сборник практик для более эффективной разработки программных проектов. Они разработали *механизм* создания и выпуска продуктов с применением автоматизации и новых инструментов, но не касались *структуры* того, как проектировать продукты, способные эволюционировать. Эволюционная архитектура рассматривает предложенные авторами инженерные практики как предпосылки для разработки эволюционирующих продуктов и описывает, как их для этого использовать.

Мы определяем эволюционную архитектуру как архитектуру, поддерживающую управляемые, инкрементные изменения в нескольких измерениях. «Инкрементный» означает, что архитектура должна меняться понемногу и постепенно. В этой главе описываются виды архитектуры, поддерживающие инкрементные изменения, а также некоторые инженерные практики, используемые для внедрения инкрементных изменений — важного структурного блока эволюционной архитектуры. Мы обсудим два направления инкрементных изменений: *разработку* — как разработчики создают программные продукты, и *эксплуатацию* — как команды развертывают созданные продукты.

В этой главе рассматриваются характеристики, методы проектирования, особенности организации команд и другие вопросы создания архитектур, поддерживающих инкрементные изменения.

[1] *Хамбл Дж., Фарли Д.* «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий».

Инкрементные изменения

Рассмотрим инкрементные изменения, затрагивающие эксплуатацию. Начнем с развернутого примера инкрементных изменений из главы 1, подробнее коснувшись деталей архитектуры и среды развертывания. У PenultimateWidgets, нашего продавца безделушек, есть сайт каталога, разработанный на основе микросервисной архитектуры и с помощью соответствующих инженерных практик, как показано на рис. 3.1.

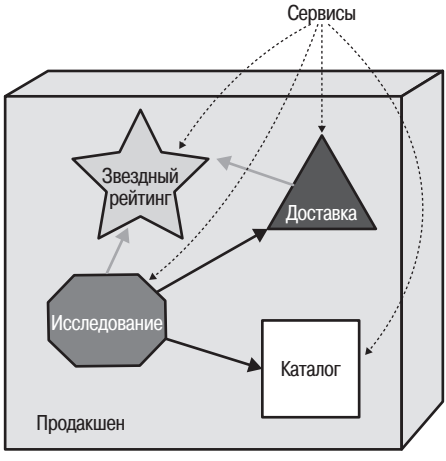


Рис. 3.1. Начальная конфигурация развертывания компонентов PenultimateWidgets

Архитекторы PenultimateWidgets реализовали микросервисы, эксплуатационно изолированные от других сервисов, с использованием архитектуры *share nothing*[1]: каждый сервис эксплуатационно обособлен, что позволяет избежать связанности и, следовательно, способствует изменениям на модульном уровне. PenultimateWidgets развертывает все свои сервисы в отдельных контейнерах, что упрощает эксплуатационные изменения.

Пользователи могут оценивать товары на сайте, ставя им звезды. Но оценивать нужно и другие части архитектуры (работу службы поддержки клиентов, поставщика услуг доставки и т. д.), поэтому все они используют сервис звездного рейтинга. Однажды команда разработчиков звездного рейтинга выпускает допол-

[1] *share nothing* — букв. «ничем не делимся», иными словами «без совместного использования» — распределенная архитектура, в которой каждый узел является независимым и самостоятельным — *Примеч. науч. ред.*

нение к его существующей версии. В новой версии добавлена возможность ставить только половину звезды — это важное обновление, как показано на рис. 3.2.

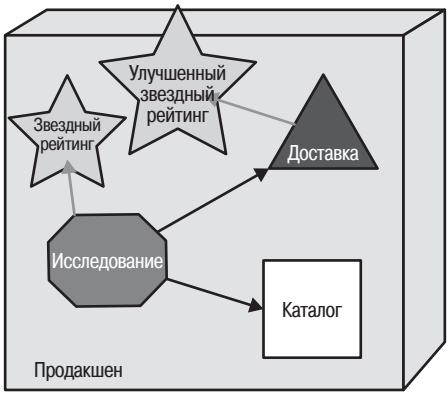


Рис. 3.2. Развертывание с улучшенным сервисом звездного рейтинга, в который добавлена возможность оценки в ползвезды

Сервисы, использующие рейтинги, не обязаны сразу переходить на новую версию, а могут постепенно мигрировать на нее в удобном для себя темпе. Со временем все больше частей экосистемы, которым нужны рейтинги, переходят на улучшенную версию. Одна из практик DevOps в PenultimateWidgets — мониторинг архитектуры, мониторинг не только сервисов, но и маршрутов между сервисами. Если эксплуатационная группа видит, что в течение определенного времени к сервису никто не обращается, она автоматически исключает его из экосистемы, как показано на рис. 3.3.

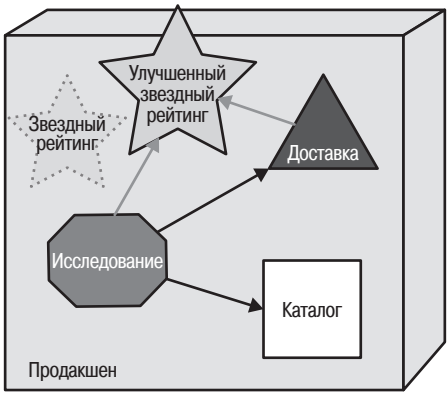


Рис. 3.3. Все сервисы теперь используют улучшенный звездный рейтинг

Механическая способность к эволюции — один из ключевых компонентов эволюционной архитектуры. Углубимся в вышеприведенную абстракцию на один уровень ниже.

PenultimateWidgets имеет мелкогранулированную архитектуру микросервисов, в которой каждый сервис развертывается с помощью контейнера — например, Docker (https://www.docker.com/) — и использует шаблон для управления связанностью инфраструктуры. Приложения в PenultimateWidgets включают пути между экземплярами запущенных сервисов — сервис может иметь несколько экземпляров для решения задач эксплуатации, таких как обеспечение масштабируемости по требованию. Это позволяет архитекторам размещать разные версии сервисов на продакшен и контролировать доступ к ним с помощью маршрутизации. Когда пайплайн развертывает службу, она регистрирует сама себя (местоположение и контракт) в инструменте обнаружения служб. Если сервису нужно найти другую службу, он использует инструмент обнаружения, чтобы узнать местоположение и версию, подходящую по контракту.

После развертывания новый сервис звездного рейтинга регистрируется в инструменте обнаружения служб и публикует свой новый контракт. Новая версия сервиса поддерживает более широкий диапазон значений — в частности, половинные значения, — чем первоначальная. Это означает, что разработчикам сервиса не нужно беспокоиться об ограничении поддерживаемых значений. Если новая версия требует другого контракта для операции вызова, то разумно решить эту проблему внутри сервиса, а не озадачивать вызывающую службу вопросом, какую версию вызывать. Мы рассмотрим эту стратегию контрактов в разделе «Выбирайте внутреннее версионирование» на с. 207 (глава 7).

При развертывании нового сервиса вызывающие службы не должны принудительно обновляться до новой версии. Поэтому архитектор временно изменяет конечную точку сервиса звездного рейтинга на прокси, который проверяет, какая версия сервиса запрашивается, и направляет к запрошенной версии. Существующие сервисы могут продолжать использовать звездный рейтинг как прежде, а для новых вызовов доступны преимущества новой версии. Существующие сервисы не обязаны обновляться и могут обращаться к первоначальной версии, пока она им необходима. Когда вызывающие службы решают использовать новое поведение, они запрашивают у конечной точки новую версию. Со временем первоначальная версия перестает использоваться, и архитектор может удалить старую версию из конечной точки, если она больше не нужна. Команда эксплуатации отвечает за обнаружение сервисов, к которым больше не поступают вызовы от других сервисов (в пределах заданного порога времени), и за сборку мусора. На рис. 3.3 показана схема эволюционного процесса; этот пример облачной эволюционной архитектуры реализован в инструменте Swabbie (https://github.com/spinnaker/swabbie).

Все изменения в этой архитектуре, включая предоставление внешних компонентов, таких как база данных, контролируются пайплайном развертывания, поэтому DevOps не приходится координировать отдельные перемещаемые части развертывания.

Определив фитнес-функции, необходимо обеспечить их своевременную оценку. Здесь на помощь приходит автоматизация. Для подобных задач часто используется *пайплайн развертывания*. С его помощью архитекторы могут определить, какие фитнес-функции выполняются, когда и с какой частотой.

Пайплайн развертывания

В книге «Continuous Delivery»[1] описан механизм пайплайна развертывания. Подобно серверу непрерывной интеграции, пайплайн развертывания «прослушивает» изменения, а затем выполняет этапы проверки, постепенно ее усложняя. В практике continuous delivery пайплайн развертывания активно используется в качестве механизма автоматизации общих задач проекта, таких как тестирование, выделение машин, развертывание и т. д. Инструменты с открытым исходным кодом, такие как GoCD (https://www.go.cd/), облегчают создание пайплайнов развертывания.

Типичный пайплайн развертывания автоматически создает среду развертывания (контейнер, например Docker (https://www.docker.com/) или специальную среду, созданную таким инструментом, как Puppet (https://puppet.com/) либо Chef (https://www.chef.io/chef)), как показано на рис. 3.4.

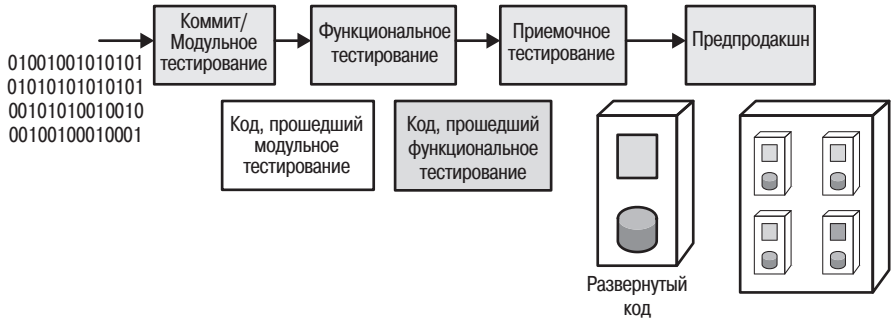


Рис. 3.4. Этапы пайплайна развертывания

[1] *Хамбл Дж., Фарли Д.* «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий».

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ И ПАЙПЛАЙНЫ РАЗВЕРТЫВАНИЯ

Непрерывная интеграция — известный метод agile-разработки, суть которого заключается в том, чтобы интегрировать как можно раньше и как можно чаще. Такие инструменты, как ThoughtWorks CruiseControl (http://cruisecontrol.sourceforge.net/) и Jenkins (https://www.jenkins.io/), а также другие коммерческие и бесплатные решения упрощают непрерывную интеграцию. Непрерывная интеграция предполагает создание выделенного места для сборки; кроме того, разработчикам удобна ее концепция единого механизма для обеспечения работоспособности кода. Вдобавок ко всему сервер непрерывной интеграции способен выделять оптимальное время и место для выполнения общих задач проекта, таких как модульное тестирование, покрытие кода, метрики, функциональное тестирование и... фитнес-функции! Для многих проектов сервер непрерывной интеграции предусматривает список задач, успешное завершение которых указывает на успех сборки. Для крупных проектов этот список довольно внушителен.

В пайплайнах развертывания удобно разбивать отдельные задачи на *этапы*. Пайплайн развертывания поддерживает концепцию многоэтапной сборки, в рамках которой разработчики могут моделировать столько задач post-check, сколько необходимо. Благодаря этой возможности разделения задач пайплайн развертывания решает более широкую задачу — проверку готовности к выпуску — по сравнению с сервером непрерывной интеграции, ориентированным в первую очередь на интеграцию. Таким образом, пайплайн развертывания обычно берет на себя тестирование приложений на нескольких уровнях, автоматическое выделение среды и множество других задач проверки.

Некоторые разработчики пытаются «обойтись» сервером непрерывной интеграции, но вскоре сталкиваются с недостаточным уровнем разделения задач и обратной связи.

Пайплайн развертывания гарантирует, что хост-компьютер (или виртуальная машина) декларативно определен и его легко перенастроить с нуля.

Пайплайн развертывания также предлагает оптимальный способ выполнения фитнес-функций, определенных для архитектуры: он применяет произвольные критерии проверки, обеспечивает поэтапное включение уровней абстракции и сложности тестов и запускается при каждом изменении системы. Пайплайн развертывания с добавленными в него фитнес-функциями приведен на рис. 3.5.

На рис. 3.5 показан набор атомарных и комплексных фитнес-функций, причем последние — в более сложной среде интеграции. Пайплайны развертывания могут обеспечить выполнение правил, определенных для защиты измерений архитектуры, при каждом изменении системы.

Рис. 3.5. Пайплайн развертывания с добавлением фитнес-функций

В главе 2 мы представили таблицу требуемых характеристик архитектуры PenultimateWidgets. Внедрив некоторые практики непрерывной доставки, разработчики поняли, что архитектурные характеристики платформы лучше работают в автоматизированном пайплайне развертывания. Поэтому они создали пайплайн развертывания для проверки фитнес-функций, созданных архитекторами системы предприятия совместно с командой разработчиков сервиса. Теперь каждый раз, когда команда вносит изменения в сервис, многочисленные тесты проверяют правильность кода и его общее соответствие архитектуре.

Другая практика, часто используемая в проектах эволюционной архитектуры, — непрерывное развертывание: применение пайплайна развертывания для внесения изменений на продакшен в зависимости от его успешного прохождения через множество тестов и других проверок. Хотя непрерывное развертывание — лучший возможный вариант, оно требует четкой координации: разработчики должны следить, чтобы изменения, постоянно внедряемые на продакшен, не вели к поломкам.

Для решения проблемы координации в пайплайнах развертывания обычно используется операция *fan-out* (разветвление по выходу), когда пайплайн выполняет несколько задач параллельно, как показано на рис. 3.6.

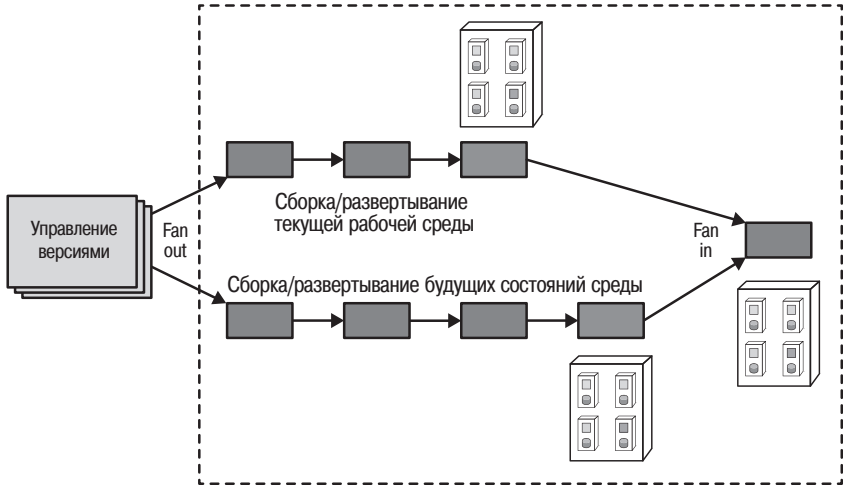


Рис. 3.6. Тестирование нескольких сценариев в пайплайне развертывания с применением разветвления по выходу

Как показано на рис. 3.6, когда команда вносит изменения, она должна обеспечить два условия: изменения не должны оказать негативного влияния на текущее состояние рабочей среды (поскольку успешное выполнение пайплайна развертывания приведет к развертыванию кода на продакшен) и должны пройти успешно (влиять на будущее состояние среды). В пайплайне развертывания с разветвлением по выходу (fan-out) задачи (тестирование, развертывание и т. д.) выполняются параллельно, что экономит время. После завершения серии параллельных задач, показанных на рис. 3.6, пайплайн оценивает результаты и, если они успешны, выполняет операцию *fan-in* (соединение по входу), объединяя их в единый поток действий для выполнения таких задач, как развертывание. Обратите внимание, что пайплайн развертывания может выполнять комбинацию *fan-out* и *fan-in* множество раз, если команде необходимо оценить изменение в нескольких контекстах.

Осуществляя непрерывное развертывание, часто приходится решать проблему его влияния на бизнес. Пользователи отрицательно относятся к постоянному появлению новых функций и предпочли бы, чтобы они добавлялись более привычным образом, например в виде развертывания «большого взрыва». Для проведения одновременно и непрерывного развертывания, и пошаговых релизов часто применяются переключатели функций (feature toggles) (https://martinfowler.com/articles/feature-toggles.html). Переключатель функций обычно представляет собой условие в коде, которое активирует или деактивирует функцию или переключает ее между двумя реализациями (например, новой и старой). Простейшей реализацией переключателя является if-выражение, которое проверяет пере-

менную среды или значение конфигурации и выводит или скрывает функцию на основе значения этой переменной. Можно использовать и более сложные переключатели функций, которые обеспечивают возможность перезагрузки конфигурации и включения или отключения функций во время выполнения. Вводя переключатели, разработчики могут безопасно развертывать изменения на продакшен, не беспокоясь, что они станут доступны пользователям раньше времени. На самом деле многие команды, выполняющие непрерывное развертывание, используют переключатели функций, чтобы отделить введение новых функций в эксплуатацию от их выпуска для пользователей.

> **QA НА ПРОДАКШЕН**
>
> Один из полезных побочных эффектов привычного создания новых функций с помощью переключателя — возможность выполнять QA на продакшен. Многие компании не знают, что могут проводить исследовательское тестирование в своей рабочей среде. Когда команда освоит использование переключателей, она сможет развертывать изменения на продакшен, поскольку большинство фреймворков для переключателей позволяет маршрутизировать пользователей по ряду критериев (IP-адрес, список контроля доступа (ACL) и др.). Если команда применяет переключатели для развертывания новых функций, доступных только отделу QA, такие функции можно тестировать на продакшен.

Используя пайплайны развертывания, архитекторы могут с легкостью применять в проекте фитнес-функции. Основная сложность для разработчиков, проектирующих пайплайн развертывания, состоит в определении необходимых этапов. Однако внедрение в пайплайн требуемых фитнес-функций значительно снижает риск того, что эволюционные изменения нарушат руководящие принципы проекта. Зачастую архитектурные проблемы остаются недостаточно раскрытыми и недооцененными, и к ним относятся субъективно; если же представить их в виде фитнес-функций, это позволит повысить точность методов разработки, выбранных для их решения, и, следовательно, уверенность в этих методах.

Практический пример: добавление фитнес-функций в сервис выставления счетов PenultimateWidgets

В архитектуре нашей вымышленной компании PenultimateWidgets существует сервис обработки счетов. Команда, отвечающая за эту функцию, хочет заменить устаревшие библиотеки и подходы, но при этом убедиться, что эти изменения не повлияют на способность других команд интегрироваться с ними.

Команда определила следующие потребности:

Масштабируемость

Хотя производительность не имеет для PenultimateWidgets решающего значения, компания обрабатывает данные счетов для нескольких реселлеров, поэтому сервис выставления счетов должен поддерживать соглашения об уровне обслуживания (SLA, Service-Level Agreement).

Интеграция с другими сервисами

Сервис выставления счетов используют и несколько других сервисов в экосистеме PenultimateWidgets. Команда хочет убедиться, что точки интеграции не будут повреждены при внесении изменений.

Безопасность

Выставление счетов подразумевает денежные операции, поэтому необходимо постоянно обеспечивать безопасность.

Проверяемость

По закону изменения, связанные с операциями налогообложения, должны проверяться независимым аудитором.

Команда использует сервер непрерывной интеграции и недавно перешла на предоставление среды, в которой выполняется код, по требованию. Чтобы реализовать фитнес-функции эволюционной архитектуры, она внедрила пайплайн развертывания вместо сервера непрерывной интеграции, что позволило выделить несколько этапов выполнения, как показано на рис. 3.7.

Пайплайн развертывания PenultimateWidgets включает шесть этапов:

Этап 1: репликация непрерывной интеграции

На первом этапе реплицируется поведение существующего сервера непрерывной интеграции, выполняются модульные и функциональные тесты.

Этап 2: контейнеризация и развертывание

Разработчики используют второй этап, чтобы создавать контейнеры для сервиса. Это позволяет проводить более глубокое тестирование, включая развертывание контейнеров в динамической тестовой среде.

Этап 3: выполнение атомарных фитнес-функций

На третьем этапе выполняются атомарные фитнес-функции, в том числе автоматизированное тестирование масштабируемости и тестирование на проникновение в систему безопасности. На этом этапе также запускается метрика, которая помечает весь код, который был изменен разработчиками

в заданном пакете, для проверки аудитором. Хотя этот инструмент ничего не определяет, он помогает в дальнейшем сузить целевую область кода.

Этап 4: выполнение комплексных фитнес-функций

На четвертом этапе выполняются комплексные фитнес-функции, в том числе тестирование контрактов для защиты точек интеграции и некоторые дополнительные тесты масштабируемости.

Этап 5a: проведение проверки безопасности (вручную)

Этот этап выполняется группой безопасности организации, которая вручную проводит обзор, аудит и оценку уязвимостей безопасности в кодовой базе. Пайплайны развертывания допускают ручное выполнение определенных этапов, запускаемое по требованию специалиста по безопасности.

Этап 5b: проведение аудитов (вручную)

Компания PenultimateWidgets расположена в штате, где действуют особые правила аудита. Команда встраивает этот ручной этап в пайплайн

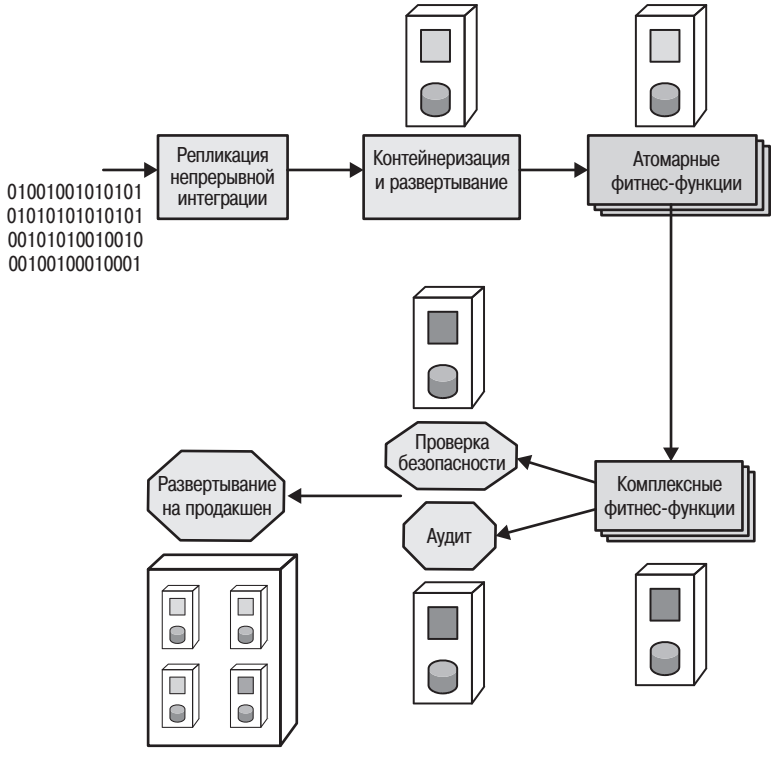


Рис. 3.7. Пайплайн развертывания PenultimateWidgets

развертывания, что дает несколько преимуществ. Во-первых, выражение аудита в виде фитнес-функции позволяет разработчикам, архитекторам, аудиторам и другим заинтересованным сторонам получать единое представление этого поведения. Это необходимо для того, чтобы убедиться, что система функционирует надлежащим образом. Во-вторых, добавление аудита в пайплайн развертывания позволяет разработчикам оценить инженерную сторону этого поведения в сравнении с другими автоматизированными тестами в пайплайне.

Например, если проверка безопасности происходит раз в неделю, а аудит только раз в месяц, то узким местом для более быстрого выпуска релизов, очевидно, является этап аудита. Если рассматривать и безопасность, и аудит как этапы пайплайна развертывания, то решения, касающиеся и того и другого, можно принимать более взвешенно: принесет ли пользу компании увеличение частоты выпуска релизов за счет того, что консультанты станут чаще проводить аудит?

Этап 6: развертывание

Последний этап — развертывание на продакшен. В PenultimateWidgets это автоматизированный этап, который запускается, только если два предыдущих ручных этапа (*проверка безопасности* и *аудит*) завершились успешно.

Архитекторы PenultimateWidgets каждую неделю получают автоматический отчет об успехах/неудачах фитнес-функций и оценивают их общее состояние, периодичность выполнения и другие факторы.

Практический пример: проверка согласованности API в автоматизированной сборке

Архитекторы PenultimateWidgets разработали API, инкапсулирующий внутреннюю сложность бухгалтерских систем предприятия в более понятный интерфейс, который используют остальные сотрудники компании (и компании-партнеры). Поскольку интегрированных потребителей много, необходимо внедрять изменения с осторожностью, чтобы не возникло расхождений или сбоев при работе с предыдущими версиями.

Для этого архитекторы разработали пайплайн развертывания, показанный на рис. 3.8.

Он состоит из пяти этапов:

Этап 1: проектирование

Артефакты проектирования, включая новые и измененные записи для API интеграции.

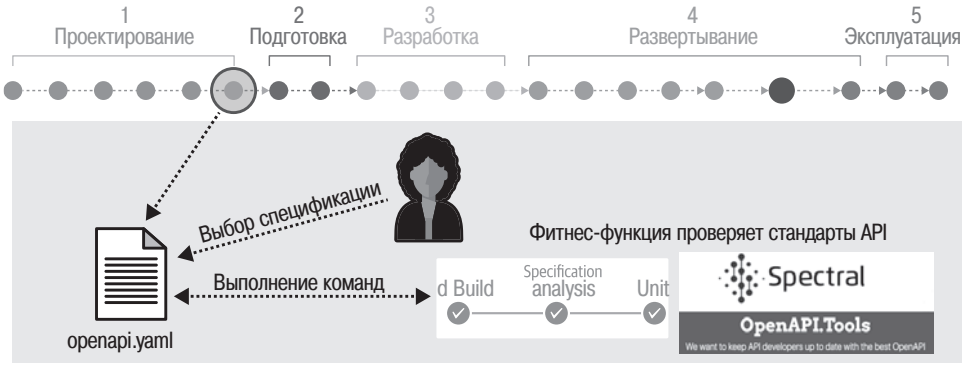


Рис. 3.8. Фитнес-функция согласованности как часть пайплайна развертывания

Этап 2: подготовка

Выполнение подготовительных задач эксплуатации, в том числе задач контейнеризации и миграции баз данных, что необходимо для проведения остальных тестов и проверок в пайплайне развертывания.

Этап 3: разработка

Разработка среды тестирования для модульного, функционального и пользовательского приемочного тестирования, а также фитнес-функций архитектуры.

Этап 4: развертывание

Если все вышеперечисленные задачи выполнены успешно, происходит развертывание на продакшен под управлением переключателя функций, который контролирует доступность новых функций для пользователей.

Этап 5: эксплуатация

Поддержка непрерывных фитнес-функций и других мониторов.

На случай изменения API архитекторы разработали многокомпонентную фитнес-функцию. Этап 1 цепочки проверки начинается с проектирования и формирования нового API, опубликованного в файле `openapi.yaml`. Команда проверяет структуру и другие параметры новой спецификации с помощью Spectral (https://stoplight.io/open-source/spectral) и OpenAPI.Tools (https://openapi.tools/).

Следующий этап в пайплайне развертывания приходится на начало разработки, как показано на рис. 3.9.

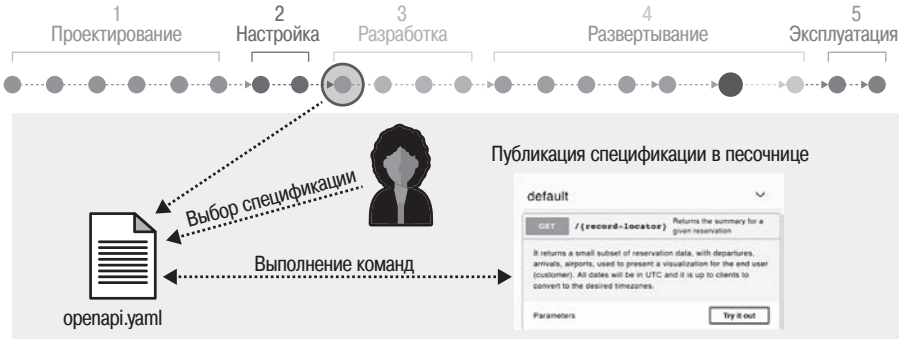


Рис. 3.9. Второй этап проверки согласованности

На этапе 2, показанном на рис. 3.9, пайплайн развертывания выбирает новую спецификацию и публикует ее в песочнице для тестирования. После запуска песочницы пайплайн выполняет серию модульных и функциональных тестов для проверки изменений. На этом этапе проверяется, что приложения, предоставляющие API, по-прежнему функционируют стабильно.

Этап 3, показанный на рис. 3.10, тестирует параметры интеграционной архитектуры с помощью Pact (https://docs.pact.io/), инструмента, который позволяет проводить кросс-сервисное интеграционное тестирование для проверки точек интеграции, что является реализацией общепринятой концепции контрактов, ориентированных на потребителя.

Контракты, ориентированные на потребителя (consumer-driven contracts, CDC) (https://martinfowler.com/articles/consumerDrivenContracts.html), представляющие собой атомарные фитнес-функции интеграционной архитектуры, часто применяются в архитектурах микросервисов. Рассмотрим рис. 3.11.

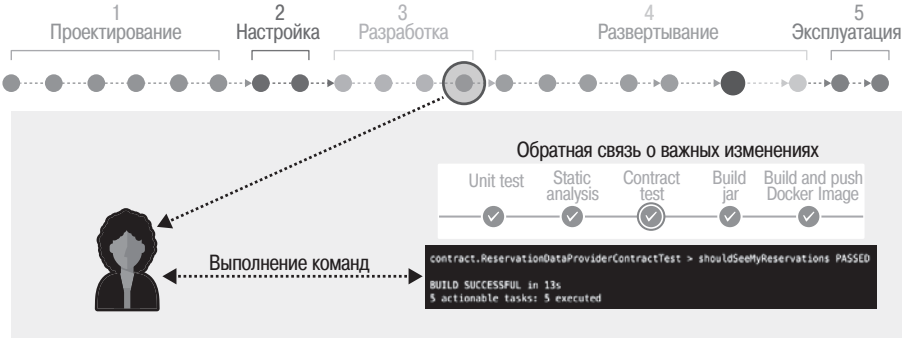


Рис. 3.10. Третий этап проверки согласованности в интеграционной архитектуре

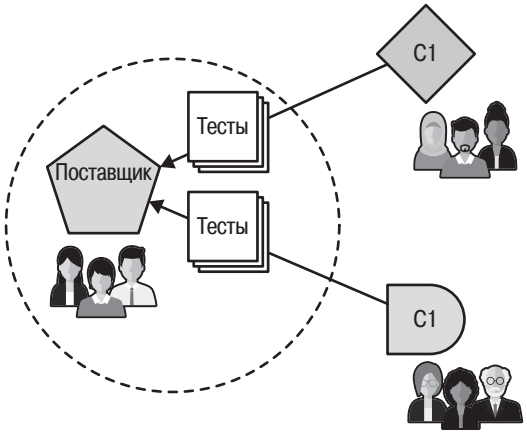


Рис. 3.11. Концепция CDC использует тесты для заключения контрактов между поставщиком и потребителем (потребителями)

На рис. 3.11 команда *поставщика* предоставляет информацию (обычно данные в легком формате, например JSON) каждому из потребителей, *C1* и *C2*. В контрактах, ориентированных на потребителя, потребители информации составляют набор тестов, инкапсулирующих требования к поставщику, и передают эти тесты поставщику, который обещает обеспечивать их постоянное успешное прохождение. Поскольку тесты включают данные, необходимые потребителю, поставщик может вносить любые изменения, которые не вызовут сбой фитнес-функций. В сценарии, показанном на рис. 3.11, *поставщик* выполняет тесты от имени всех трех потребителей в дополнение к своему собственному набору тестов. Для подобного использования фитнес-функций существует неформальный термин *страховочная сеть* (engineering safety net). Не стоит заниматься поддержкой согласованности протоколов интеграции вручную, если можно легко построить фитнес-функцию и поручить ей эту работу.

При инкрементных изменениях эволюционной архитектуры неявно подразумевается определенный уровень инженерной зрелости команд разработчиков. Например, если команда использует контракты, ориентированные на потребителя, но при этом несколько дней подряд выпускает нерабочие сборки, она не может быть уверена, что ее точки интеграции актуальны. Использование инженерной практики для контроля процессов с помощью фитнес-функций избавляет разработчиков от большого количества ручных операций, но это требует достаточного уровня зрелости.

На последнем этапе пайплайн развертывания передает изменения на продакшен, чтобы перед официальным запуском можно было провести A/B-тестирование и другие проверки.

Итоги

Некоторые из нас (авторов) работали в инженерных сферах, не связанных с программированием, как, например, Нил, который начал свое университетское образование с традиционных инженерных дисциплин. Прежде чем перейти к разработке, он усвоил хорошую порцию высшей математики, которую применяют инженеры-конструкторы.

Довольно часто разработку программного обеспечения пытаются приравнять к другим инженерным дисциплинам. Например, применять в разработке ПО *водопадный процесс* проектирования всей системы с последующей механической сборкой, что оказалось совершенно для нее не подходящим. Другой вопрос, который часто задают разработчикам: когда же вы будете использовать такую же математику, как традиционные инженеры?

Маловероятно, что в *программной инженерии* математика будет когда-либо использоваться так же широко, как в других инженерных дисциплинах, поскольку *проектирование* и *производство* — далеко не одно и то же. В строительной инженерии производство обходится очень дорого, что делает недостатки конструкции недопустимыми, и, чтобы их избежать, на этапе проектирования приходится выполнять тщательный и всеобъемлющий прогностический анализ. Таким образом, усилия между проектированием и производством распределяются равномерно. В программной инженерии, однако, баланс совершенно иной. Производство программных продуктов равнозначно их компиляции и развертыванию — действиям, которые все чаще автоматизируются. Таким образом, в программной инженерии практически все усилия приходятся на проектирование, а не на производство; проектирование включает в себя написание кода и вообще практически все, что мы называем разработкой программного обеспечения.

Однако продукты, которые мы производим, также значительно отличаются друг от друга. Современное ПО состоит из тысяч или миллионов компонентов, которые можно изменять буквально как угодно. К счастью, разработчики могут вносить эти изменения в дизайн системы и затем практически мгновенно развернуть (по сути, заново произвести) ее.

В основе настоящей программной инженерии лежат инкрементные изменения и их автоматизированные проверки. Поскольку мы можем производить по сути что угодно, но при этом наши продукты чрезвычайно изменчивы, секрет успешной разработки в том, чтобы с уверенностью, основанной на автоматизированных проверках, вносить изменения — инкрементные изменения.

ГЛАВА 4

Автоматизация управления архитектурой

В задачи архитекторов входит создание дизайна структуры программных систем, а также определение методов разработки и проектирования. Однако еще одной важной задачей архитекторов является *управление* вопросами создания ПО, в том числе принципами проектирования, лучшими практиками и выявлением подводных камней, которых следует избегать.

В прошлом, чтобы устанавливать политики управления, архитекторы в основном были вынуждены ограничиваться ручными проверками кода, советами по архитектуре и другими неэффективными средствами. Однако с появлением автоматизированных фитнес-функций у нас возникли новые возможности. В этой главе мы опишем, как для создания автоматизированных политик управления архитекторы могут использовать фитнес-функции, предназначенные для эволюции программных продуктов.

Фитнес-функции для управления архитектурой

Эта книга возникла из идеи объединения программной архитектуры и практики разработки генетических алгоритмов (о чем говорилось в главе 2) с акцентом на создании проектов, которые со временем должны успешно развиваться, а не деградировать. Результатом развития этой идеи стали приведенные здесь различные способы описания и применения фитнес-функций.

Однако, хотя это и не было частью первоначального замысла, мы поняли, что механика эволюционной архитектуры в значительной степени пересекается с *управлением архитектурой*, особенно с принципом *автоматизации* управления, которая сама по себе представляет эволюцию методов программной инженерии.

В начале 1990-х годов Кент Бек (Kent Beck) возглавил группу перспективных разработчиков, которые раскрыли одну из движущих сил прогресса программной инженерии, случившегося за последние три десятилетия. Они работали над проектом C3 (его предмет не представляет важности). Команда была хорошо знакома с современными тенденциями в разработке, но они их не впечатляли — казалось, что ни один из популярных в то время методов не приносит стабильно успешного результата. Тогда Кент выдвинул идею экстремального программирования (eXtreme Programming, XP (http://www.extremeprogramming.org/)): он и его команда взяли то, что, как они знали из прошлого опыта, работает, и выполнили это еще раз, но максимально экстремальным способом. Например, их опыт показал, что в проектах с более высоким покрытием тестами код, как правило, более качественный, поэтому они начали внедрять разработку на основе тестов, которые покрывали бы весь код, поскольку их писали перед тем, как писать код.

Одно из их ключевых наблюдений касалось интеграции. В то время в большинстве программных проектов было принято проводить *фазу интеграции*. Разработчики должны были по отдельности писать код в течение нескольких недель или месяцев, а затем на этапе интеграции объединять свои версии. Фактически подобная изоляция на уровне разработчика была вынужденной из-за специфики многих популярных в то время инструментов контроля версий. Введение фазы интеграции было обусловлено многочисленными и зачастую неверно применяемыми к разработке аналогиями из производственной сферы. XP-разработчики обратили внимание на то, что чем чаще проводится интеграция, тем меньше ошибок в продукте, что привело их к идее непрерывной интеграции: каждый разработчик должен делать коммиты основной ветки по крайней мере раз в день.

Непрерывная интеграция и многие другие практики XP иллюстрируют силу автоматизации и инкрементных изменений. Команды, использующие непрерывную интеграцию, тратят меньше времени не только на регулярные задачи слияния, но и меньше времени в целом. При проведении непрерывной интеграции конфликты слияния возникают и разрешаются так же быстро, как и появляются, по крайней мере, раз в день. В проектах, предусматривающих фазу окончательной интеграции, конфликты слияния, комбинируясь, образуют большие комки грязи, которые приходится разбивать в конце проекта.

Автоматизация важна не только для интеграции; она также выступает оптимизатором разработки. До появления непрерывной интеграции разработчики были вынуждены снова и снова тратить время на выполнение ручных задач (интеграции и слияния); непрерывная интеграция (и связанная с ней периодичность) автоматизировала большую часть этой неблагодарной работы.

Мы заново открыли преимущества автоматизации в начале 2000-х годов в ходе революции DevOps. Команды были заняты установкой операционных систем, применением исправлений и другими ручными операциями, а важные пробле-

мы оставались незамеченными. Появление автоматического резервирования машин с помощью таких инструментов, как Puppet (https://puppet.com/) и Chef (https://www.chef.io/), позволило автоматизировать инфраструктуру и обеспечить ее согласованность.

Во многих организациях мы наблюдали те же неэффективные ручные процессы в архитектуре: архитекторы пытались выполнять проверки управления, проводя анализ кода, вводя советы по архитектуре и прочие ручные бюрократизированные процессы, а важное упускали из виду. Привязав фитнес-функции к непрерывной интеграции, архитекторы могут преобразовать метрики и другие проверки управления в единую проверку целостности, проводимую на регулярной основе.

Во многих отношениях объединение фитнес-функций и инкрементных изменений под знаком непрерывной интеграции представляет собой эволюцию инженерных практик. Подобно тому как команды используют инкрементные изменения для интеграции и DevOps, мы все чаще видим, как те же принципы применяются и к управлению архитектурой.

Фитнес-функции существуют для каждого уровня архитектуры, от низкоуровневого анализа на основе кода до архитектуры системы всего предприятия. Наши примеры автоматизации управления архитектурой построены таким же образом, начиная с уровня кода, а затем охватывая весь стек разработки. Мы рассмотрим несколько фитнес-функций; на рис. 4.1 представлена наша дорожная карта.

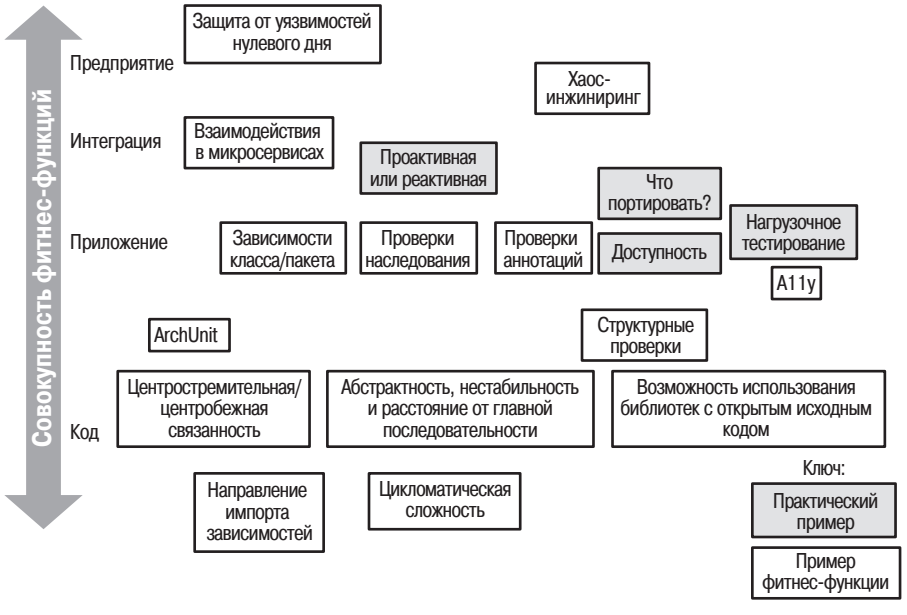


Рис. 4.1. Обзор фитнес-функций

Начнем с нижней части карты на рис. 4.1 — с фитнес-функций на основе кода, — и постепенно будем продвигаться к вершине.

Фитнес-функции на основе кода

Программные архитекторы с изрядной долей зависти относятся к другим инженерным дисциплинам, в которых существуют многочисленные методы анализа, позволяющие определить, как будут работать их проекты. У нас (пока) и близко нет инженерной математики такого уровня глубины и сложности, особенно в области архитектурного анализа.

Однако несколько подходящих инструментов, как правило, основанных на метриках на уровне кода, у нас есть. Следующие несколько разделов посвящены разбору метрик, определяющих важные компоненты архитектуры.

Центростремительная и центробежная связанность

В 1979 году Эдвард Йордон (Edward Yourdon) и Ларри Константин (Larry Constantine) опубликовали книгу «Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design» (издательство Prentice-Hall), в которой дали определение многих основных понятий разработки, включая метрики центростремительной, или афферентной (afferent), и центробежной, или эфферентной, (efferent) связанности. *Центростремительная* связанность измеряет количество *входящих* соединений с артефактом кода (компонентом, классом, функцией и т. д.). *Центробежная* — количество *исходящих* соединений с другими артефактами кода.

Связанность представляет интерес для архитекторов, поскольку она ограничивает многие другие характеристики архитектуры и влияет на них. Если позволить компонентам соединяться между собой неуправляемым образом, чаще всего в кодовой базе возникнет множество связей, не поддающихся пониманию. На рис. 4.2 показан вывод метрик реально существующей программной системы (ее название по понятным причинам не приводим).

На рис. 4.2 компоненты показаны в виде отдельных точек по периметру, а связи между компонентами — в виде линий, где жирность линии указывает на силу связи. Это пример плохой кодовой базы — изменения в любом ее компоненте могут повлиять на другие компоненты.

Практически на каждой платформе предусмотрены инструменты для анализа характеристик связанности кода, помогающие архитекторам понять и реструктурировать кодовую базу, а также проводить ее миграцию. На разных платформах существуют также инструменты, представляющие связи классов и/или компонентов в виде матрицы, как показано на рис. 4.3.

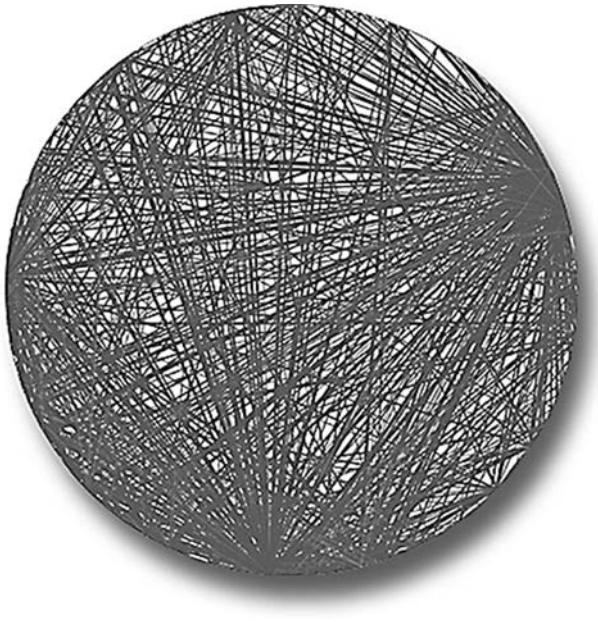


Рис. 4.2. Связанность на уровне компонентов в большом комке грязи (big ball of mud)

На рис. 4.3 показано табличное представление результатов работы в плагине Jdepend для Eclipse, включающих анализ связей в каждом пакете, а также некоторые агрегированные метрики, о которых речь пойдет в следующем разделе.

Эта и многие другие метрики, которые мы обсудим, доступны в ряде других инструментов. В частности, IntelliJ (https://www.jetbrains.com/idea) для Java, Sonar Qube (https://www.sonarqube.org/), JArchitect (https://www.jarchitect.com/) и т. д., в зависимости от выбранной платформы или технологического стека. Например, в IntelliJ можно сформировать матрицу структуры зависимостей, визуализирующую различные характеристики связанности, см. рис. 4.4.

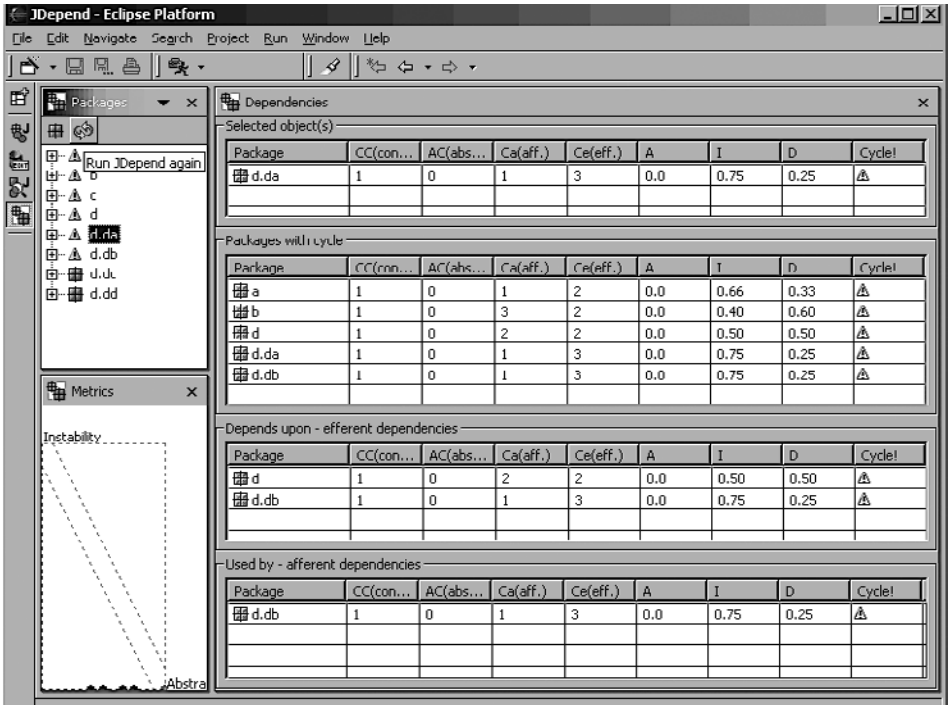


Рис. 4.3. Представление результатов анализа связанности JDepend в Eclipse

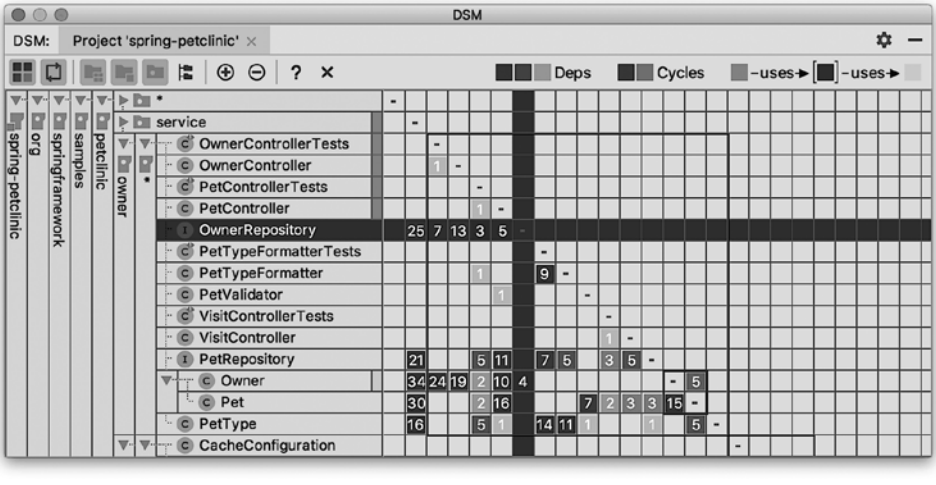


Рис. 4.4. Матрица структуры зависимостей в IntelliJ

Абстрактность, нестабильность и расстояние от главной последовательности

Роберт Мартин (Robert Martin), известная фигура в мире программной архитектуры, в конце 1990-х годов разработал несколько производных метрик, применимых к любому объектно-ориентированному языку. Эти метрики — абстрактность (abstractness) и нестабильность (instability) — измеряют отношение внутренних характеристик кодовой базы.

Абстрактность — это соотношение между абстрактными артефактами (абстрактными классами, интерфейсами и т. д.) и конкретными артефактами (классами реализации). Она представляет собой меру отношения *абстракции* и *реализации*. Абстрактные элементы кодовой базы помогают разработчикам лучше понять общее назначение кода. Например, кодовая база, состоящая из одного метода `main()` и 10 000 строк кода, получит почти нулевой балл абстракции и будет довольно сложной для понимания.

Формула абстрактности представлена в уравнении 4.1.

Уравнение 4.1. Абстрактность

$$A = \frac{\sum m_a}{\sum m_c + \sum m_a}$$

В этом уравнении m_a представляет *абстрактные* элементы (интерфейсы или абстрактные классы) кодовой базы, а m_c — *конкретные* (concrete) элементы. Архитекторы рассчитывают *абстрактность*, вычисляя отношение суммы абстрактных артефактов к сумме конкретных.

Другая производная метрика, *нестабильность*, представляет собой отношение величины центробежной связанности к сумме величин центробежной и центростремительной связанности, показанное в уравнении 4.2.

Уравнение 4.2. Нестабильность

$$I = \frac{C_e}{C_e + C_a}$$

В этом уравнении C_e — *центробежная* (или исходящая) связанность, а C_a — *центростремительная* (или входящая) связанность.

Метрика *нестабильности* определяет изменчивость кодовой базы. Кодовая база с высокой степенью нестабильности легче ломается при изменениях из-за высокой степени связанности. Рассмотрим два сценария, в каждом из которых C_a

равно 2. В первом сценарии $C_e = 0$, что обусловливает нулевую нестабильность. Во втором сценарии $C_e = 3$, что дает показатель нестабильности 3/5. Таким образом, показатель нестабильности компонента отражает, сколько потенциальных изменений могут вызвать изменения в смежных компонентах. Компонент со значением нестабильности, близким к 1, очень нестабилен, а со значением, близким к 0, может быть либо стабильным, либо жестким: стабильным, если модуль или компонент содержит в основном абстрактные элементы, и жестким, если он состоит в основном из конкретных элементов. Компромиссным условием высокой стабильности является невозможность повторного использования — если каждый компонент независим, то его можно дублировать с высокой степенью вероятности.

Компонент со значением *I*, близким к 1, очень нестабилен. Однако компонент со значением *I*, близким к 0, может быть как стабильным, так и жестким. Если он содержит в основном конкретные элементы, то является жестким.

Таким образом, важно рассматривать значения *I* и *A* в совокупности, а не по отдельности; они объединены в следующей метрике, *расстоянии от главной последовательности* (distance from the main sequence).

Одной из немногих комплексных метрик структуры является *нормализованное расстояние от главной последовательности* — производная метрика, основанная на *нестабильности* и *абстрактности* (уравнение 4.3).

Уравнение 4.3. Нормализованное расстояние от главной последовательности

$$D = |A + I - 1|$$

В уравнении A = *абстрактность*, а I = *нестабильность*.

Нормализованное расстояние от главной последовательности выражает идеальное соотношение абстрактности и нестабильности; в компонентах, расположенных вблизи такой идеальной линии, соотношение этих двух конкурирующих метрик оптимальное. Например, чтобы рассчитать для выбранного компонента его *расстояние от главной последовательности,* полезно построить график для этого компонента, как показано на рис. 4.5.

Разработчики строят график выбранного компонента (рис. 4.5), затем измеряют расстояние от идеальной линии. Чем оно меньше, тем сбалансированнее компонент. Компоненты, находящиеся далеко в правом верхнем углу, попадают в зону, которую архитекторы называют *зоной бесполезности* (zone of uselessness): слишком абстрактный код становится трудно использовать. И наоборот, код в левом нижнем углу попадает в *зону мучений* (zone of pain): это код, в котором слишком много реализации и недостаточно абстракции, его легко сломать и трудно обслуживать (рис. 4.6).

Рис. 4.5. Нормализованное расстояние от главной последовательности для выбранного компонента

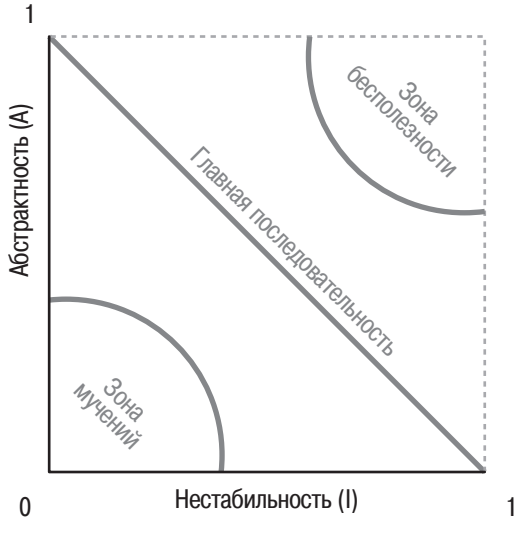


Рис. 4.6. Зона бесполезности и зона мучений

Такой анализ полезен для оценки (например, при миграции от одного архитектурного стиля к другому) либо для задания фитнес-функции. Рассмотрим скриншот на рис. 4.7, демонстрирующий использование коммерческого инструмента

NDepend (https://www.ndepend.com/) применительно к открытому инструменту тестирования NUnit (https://nunit.org/).

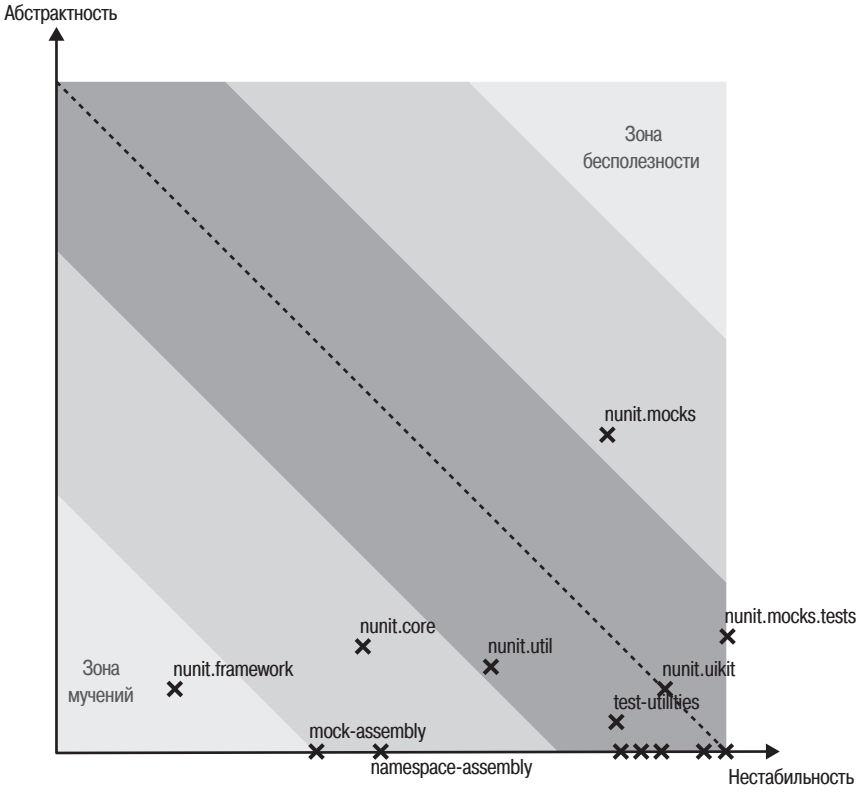


Рис. 4.7. Расстояние от главной последовательности, вычисленное в инструменте NDepend для библиотеки тестирования NUnit

На рис. 4.7 видно, что бóльшая часть кода находится рядом с линией *главной последовательности*. Компоненты `mocks` стремятся к *зоне бесполезности*: они довольно абстрактны и нестабильны. Это нормально для компонентов `mock`, которые, как правило, работают косвенно. Беспокойство вызывает то, что код `framework` соскользнул в *зону мучений*: в нем явно не хватает абстрактности и нестабильности. Что из себя представляет такой код? Множество раздутых методов, которые почти не предполагают повторного использования.

Как архитектор может приблизить проблемный код к линии главной последовательности? С помощью инструментов рефакторинга в IDE: найти крупные методы, обусловливающие эту метрику, и извлечь из них какие-то части, повы-

шая абстрактность. В ходе этой работы обнаружится дублирование извлеченного кода, который можно удалить и тем самым повысить нестабильность.

Прежде чем проводить рефакторинг и перемещать кодовую базу, необходимо проанализировать и оптимизировать ее с использованием описанных выше метрик. Как и в обычном строительстве, перенести конструкцию с неустойчивым основанием сложнее, чем с прочным.

Эту метрику также можно использовать в качестве фитнес-функции, чтобы убедиться, что качество кодовой базы не снизилось до недопустимого.

Направленность импорта

В ситуации, подобной показанной на рис. 2.3, важно контролировать направленность импорта. В экосистеме Java JDepend (https://github.com/clarkware/jdepend) это инструмент, который анализирует характеристики связанности пакетов. Поскольку JDepend написан на Java, в нем есть API, который разработчики могут использовать для проведения собственного анализа с помощью модульных тестов.

Рассмотрим фитнес-функцию в примере 4.1, выраженную в виде теста JUnit (http://junit.org/).

Пример 4.1. Тест JDepend для проверки направленности импорта пакетов

```
public void testMatch() {
    DependencyConstraint constraint = new DependencyConstraint();

    JavaPackage persistence = constraint.addPackage("com.xyz.persistence");
    JavaPackage web = constraint.addPackage("com.xyz.web");
    JavaPackage util = constraint.addPackage("com.xyz.util");

    persistence.dependsUpon(util);
    web.dependsUpon(util);

    jdepend.analyze();

    assertEquals("Dependency mismatch", true,
            jdepend.dependencyMatch(constraint));
}
```

В примере 4.1 мы сначала определяем пакеты в приложении, а затем задаем правила импорта. Если разработчик случайно напишет код, который импортирует в `util` из `persistence`, модульный тест завершится неудачей до

коммита кода. Лучше создавать модульные тесты, выявляющие нарушения архитектуры, а не использовать строгие правила разработки (с сопутствующей бюрократией): это позволит разработчикам сосредоточиться на предметной области и меньше отвлекаться на проблемы инфраструктуры. И, что еще более важно, так архитекторы смогут консолидировать правила в виде исполняемых артефактов.

Цикломатическая сложность и управление в стиле «пасти»[1]

Часто используемой метрикой кода является цикломатическая сложность — мера измерения сложности кода, доступная для всех структурированных языков программирования и применяемая уже несколько десятилетий.

Цикломатическая сложность (cyclomatic complexity, CC) — это метрика, предназначенная для объективного измерения сложности кода на уровне функций/методов, классов или приложений, разработанная Томасом Маккейбом-старшим (Thomas McCabe Sr.) в 1976 году.

Она вычисляется путем применения теории графов к коду, а именно к точкам принятия решений, где возникают различные пути выполнения. Например, если функция не содержит операторов принятия решений (таких как операторы `if`), то `CC = 1`. Если функция имеет одно условие, то `CC = 2`, поскольку существует два возможных пути выполнения.

Формула цикломатической сложности для одной функции или метода имеет вид $CC = E - N + 2$, где `N` представляет *узлы* (строки кода), а `E` – *ребра* (возможные решения). Рассмотрим C-подобный код, показанный в примере 4.2.

Пример 4.2. Пример кода для оценки цикломатической сложности

```
public void decision(int c1, int c2) {
    if (c1 < 100)
        return 0;
    else if (c1 + C2 > 500)
       return 1;
    else
       return -1;
}
```

[1] Отсылка к книге Дж.Рейнвотера «Как пасти котов» (СПб., издательство «Питер»). Смысл в том, что следует управлять не четкими приказами, а вести разработчиков по некоторой колее. — *Примеч. науч. ред.*

Цикломатическая сложность в примере 4.2 равна 3 (= 3 – 2 + 2); граф представлен на рис. 4.8.

Число 2, появляющееся в формуле цикломатической сложности, представляет собой упрощение для одной функции/метода. Общая формула для обращений по выходу к другим методам (в теории графов известным как *связанные компоненты*) имеет вид $CC = E - N + 2P$, где P — количество связанных компонентов.

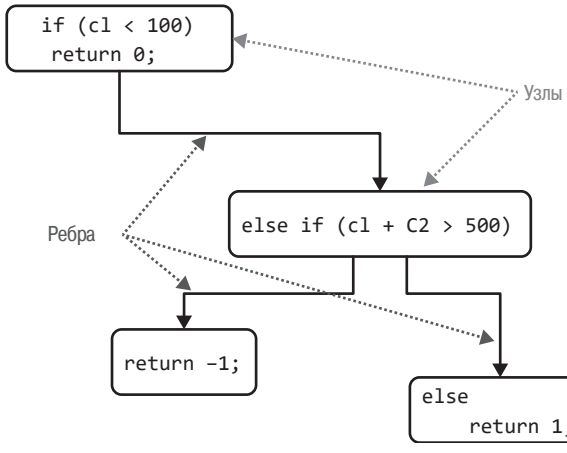


Рис. 4.8. Цикломатическая сложность для функции принятия решений

И архитекторы, и разработчики согласны с тем, что излишне сложный код — это код «с душком»; он плох практически по всем важным показателям: модульности, тестируемости, развертываемости и др. Тем не менее, если не контролировать постепенно возрастающую сложность, вскоре она начнет доминировать в кодовой базе.

CC — хороший пример метрики, которой полезно управлять; от излишней сложности кода не выиграет никто. Однако что происходит в проектах, где этот показатель долгое время игнорируется?

Вместо того чтобы устанавливать жесткий порог допустимого значения фитнес-функции, можно *направлять и подводить* разработчиков к лучшим значениям. Допустим, вы задали верхний предел допустимого значения CC равным 10, но после этого большинство проектов провалилось. Не отчаивайтесь — создайте каскадную фитнес-функцию, которая будет выдавать предупреждение при превышении заданного порога, позднее перерастающее в ошибку. Это даст командам время поэтапно погасить технический долг контролируемым образом.

КАКОЕ ЗНАЧЕНИЕ ЦИКЛОМАТИЧЕСКОЙ СЛОЖНОСТИ МОЖНО СЧИТАТЬ ПРИЕМЛЕМЫМ?

Частый вопрос, который встает при обсуждении этой темы, — каково приемлемое пороговое значение CC? Однозначного ответа на него, как и на остальные вопросы, касающиеся программной архитектуры, нет. В частности, это значение может зависеть от сложности предметной области. Одним из недостатков подобных метрик является неспособность разделить *существенную* и *случайную* сложность. Например, если у вас алгоритмически сложная задача, то ее решение будет содержать сложные функции. Архитекторам необходимо определить, чем обусловлена эта сложность — предметной областью или плохим кодом, а может быть, тем, что код плохо разделен. Другими словами, возможно ли разбить крупный метод на мелкие логичные фрагменты, тем самым распределив работу (и сложность) между более удачно организованными методами?

В целом для простой предметной области, а также если нет иных особых условий, в качестве приемлемых приняты значения ниже 10. Мы считаем этот порог завышенным и стремимся к значениям ниже 5, означающим, что код связный (cohesive) и хорошо отлаженный. Java-инструмент Crap4j (http://www.crap4j.org/) определяет качество кода, оценивая совокупность метрик CC и покрытия кода; если CC превышает 50, никакое покрытие не спасет код. Самым ужасным артефактом, встретившимся Нилу, была лежавшая в основе коммерческого продукта функция на языке C, чья CC превышала 800! Она единственная содержала более чем 4000 строк кода с операторами `GOTO` (для выхода из очень глубоко вложенных циклов).

У таких практик, как разработка на основе тестов (test-driven development, TDD), есть неожиданный (но положительный) побочный эффект, заключающийся в том, что в среднем для предметной области создаются более компактные и менее сложные методы. Разработчики стараются писать простые тесты и как можно меньше кода для их прохождения. Благодаря такому стремлению обеспечить дискретное поведение и разумные объемы тестов создаются хорошо отлаженные методы с высокой связностью и низкой CC.

Поэтапное сужение фитнес-функций, основанных на метриках, до желаемых значений позволяет командам избавиться от существующего технического долга и, кроме того, оставив фитнес-функции работать, предотвратить дальнейшее ухудшение показателей. В этом заключается принцип предотвращения деградации данных (bit rot) посредством управления.

Готовые инструменты

Поскольку все архитектуры отличаются друг от друга, трудно найти готовые инструменты для решения сложных проблем. Однако чем более распространена

экосистема, тем выше шанс подобрать подходящие, до некоторой степени общие инструменты. Вот несколько примеров.

Возможность использования библиотек с открытым исходным кодом

Компания PenultimateWidgets работала над проектом, который содержал несколько запатентованных собственных алгоритмов, а также несколько библиотек и фреймворков с открытым исходным кодом. Юристов обеспокоило, что команда разработчиков случайно использовала библиотеку, лицензия которой требовала распространения продукта, созданного с применением этой библиотеки, также на условиях свободной лицензии, чего PenultimateWidgets делать явно не хотела.

Итак, архитекторы собрали все лицензии зависимостей, а юристы их утвердили. Затем один из юристов задал щекотливый вопрос: что, если с одним из обновлений какая-нибудь из этих зависимостей обновит и условия лицензирования? Это был хороший юрист, и он напомнил, как подобное уже происходило с некоторыми библиотеками пользовательского интерфейса. Как разработчикам быть уверенными, что ни одна из библиотек не обновит лицензию без их ведома?

В первую очередь архитекторам следует поискать подходящий инструмент, проверяющий соблюдение условий лицензирования; на момент написания данной книги этой цели служит инструмент Black Duck[1]. Однако в то время архитекторы PenultimateWidgets не нашли подходящего инструмента.

Следовательно, они построили фитнес-функцию, которая работает следующим образом:

1. Отмечает в базе данных расположение каждого файла лицензии в пакете загрузки с открытым исходным кодом.
2. Сохраняет содержимое (или хеш) всего файла лицензии вместе с версией библиотеки.
3. При обнаружении новой версии открывает пакет загрузки, извлекает файл лицензии и сравнивает его с текущей сохраненной версией.
4. Если версии (или хеш) не совпадают, отмечает сборку как проваленную и отправляет уведомление юристам.

[1] https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html?intcmp=ref-bds

Обратите внимание, что данная функция не оценивает разницу между версиями библиотек и отнюдь не представляет собой какой-нибудь мощный аналитический инструмент на основе искусственного интеллекта. Как обычная фитнес-функция, она посылает уведомлениие о неожиданных изменениях. Это пример одновременно *автоматизированной* и *ручной* фитнес-функции: обнаружение изменений было автоматизировано, но отреагировать на изменения — одобрить измененную библиотеку — юристу по-прежнему необходимо вручную.

A11y и другие поддерживаемые характеристики архитектуры

Иногда к выбору нужного инструмента приводит знание того, что искать. A11y — это сокращение, обозначающее *доступность (accessibility):* производное от *a*, 11 букв и *y*. Этот параметр показывает, в какой степени приложение доступно людям с разными (в том числе ограниченными) возможностями.

Поскольку доступность — требование многих компаний и государственных учреждений, появилось большое разнообразие инструментов для проверки этой характеристики архитектуры, в том числе Pa11y (https://pa11y.org/), который позволяет выполнить сканирование статических веб-элементов в командной строке.

ArchUnit

ArchUnit — это инструмент тестирования, вдохновленный фреймворком JUnit и использующий некоторые его вспомогательные функции. Однако он предназначен для тестирования особенностей архитектуры, а не общей структуры кода. Мы уже показывали пример фитнес-функции ArchUnit на рис. 2.3; приведем еще несколько примеров возможных способов управления.

Зависимости пакетов

Пакеты разграничивают компоненты в экосистеме Java, и архитекторам часто требуется знать, как правильно «соединять» пакеты. Рассмотрим примеры компонентов на рис. 4.9.

Рис. 4.9. Декларативные зависимости пакетов в Java

Код ArchUnit, обеспечивающий выполнение зависимостей, показанных на рис. 4.9, представлен в примере 4.3.

Пример 4.3. Управление зависимостями пакетов

```
noClasses().that().resideInAPackage("...source...")
    .should().dependOnClassesThat().resideInAPackage("..foo.")
```

ArchUnit использует матчеры Hamcrest (https://hamcrest.org/JavaHamcrest/tutorial) для JUnit, чтобы архитекторам было проще писать более читаемые утверждения, как показано в примере 4.3, и определять, какие компоненты могут или не могут обращаться к другим компонентам.

Еще одна распространенная архитектурная проблема управления — это зависимости компонентов, см. рис. 4.10.

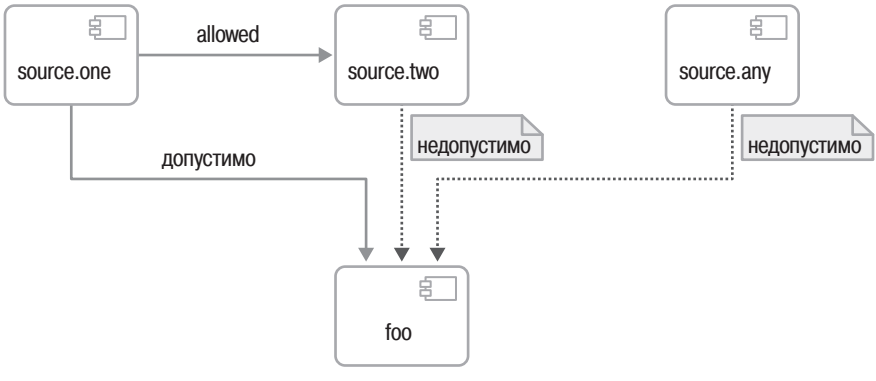


Рис. 4.10. Управление зависимостями пакетов

На рис. 4.10 общая библиотека *foo* должна быть доступна только из компонента `source.one`; архитектор может задать правило управления через ArchUnit, как в примере 4.4.

Пример 4.4. Разрешение и ограничение доступа к пакетам

```
classes().that().resideInAPackage("..foo.")
    .should().onlyHaveDependentClassesThat()
        .resideInAnyPackage("..source.one...", "..foo...")
```

Пример 4.4 показывает, как архитекторы могут контролировать зависимости между проектами во время компиляции.

Проверка зависимостей классов

Кроме пакетных зависимостей, архитекторам часто необходимо контролировать архитектурные аспекты проектирования классов. Например, ограничивать зависимости между компонентами, чтобы избежать сложностей при развертывании. Рассмотрим отношения между классами на рис. 4.11.

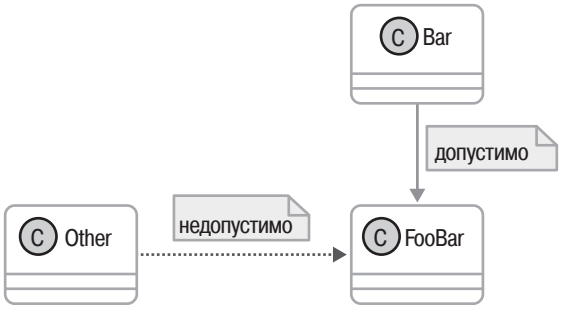


Рис. 4.11. Проверки зависимостей, разрешающие и запрещающие доступ

Правила, показанные на рис. 4.11, в ArchUnit можно записать как в примере 4.5.

Пример 4.5. Правила зависимости классов в ArchUnit

```
classes().that().haveNameMatching(".*Bar")
    .should().onlyHaveDependentClassesThat().haveSimpleName("Bar")
```

ArchUnit позволяет архитекторам контролировать все связи компонентов в приложении.

Проверки наследования

Еще одна зависимость, поддерживаемая объектно-ориентированными языками программирования, — это наследование; с точки зрения архитектуры это особая форма связанности. Станет ли оно источником проблем, зависит от того, как команды развертывают наследуемые компоненты: если наследование содержится только в одном компоненте, оно не будет иметь побочных эффектов для архитектуры. С другой стороны, если наследование выходит за границы компонентов и/или развертывания, архитекторы должны убедиться, что связанность не нарушается.

Наследование часто создает сложности для архитектуры; пример структуры, требующей управления, приведен на рис. 4.12.

Код для правил, показанных на рис. 4.12, приведен в примере 4.6.

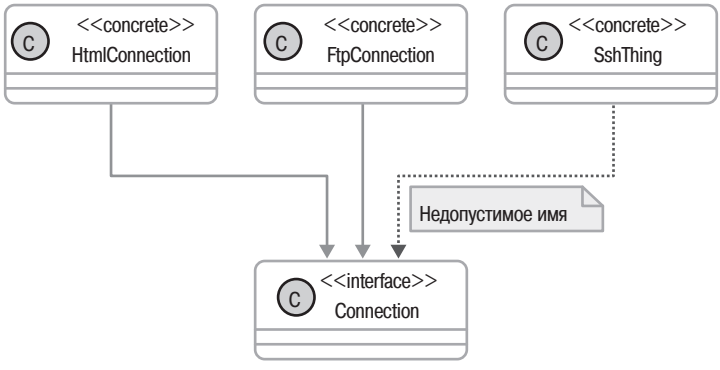


Рис. 4.12. Управление зависимостями при наследовании

Пример 4.6. Правило управления наследованием, записанное в ArchUnit

```
classes().that().implement(Connection.class)
    .should().haveSimpleNameEndingWith("Connection")
```

Проверки аннотаций

Для указания намерений в поддерживаемых платформах архитекторы обычно используют *маркерные аннотации* (tagging annotations) (или атрибуты, в зависимости от платформы). Например, архитектор планирует, что определенный класс будет действовать только как оркестратор для других сервисов — намерение состоит в том, чтобы класс никогда не принимал поведение, не связанное с оркестрацией. Добавление аннотации позволяет архитектору проверить намерение и правильность использования.

В ArchUnit можно проверить правильность использования, как показано на рис. 4.13.

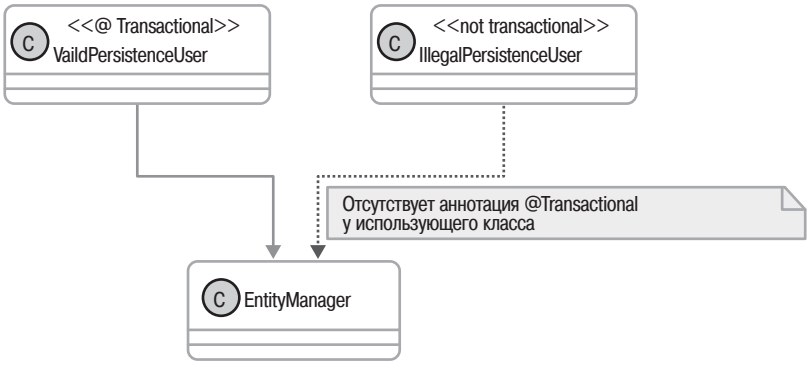


Рис. 4.13. Проверка правильного использования аннотаций

Код для правил, показанных на рис. 4.13, приведен в примере 4.7.

Пример 4.7. Правила управления аннотациями

```
classes().that().areAssignableTo(EntityManager.class)
    .should().onlyHaveDependentClassesThat().areAnnotatedWith(Transactional.
class)
```

Пример 4.7 проверяет, что только аннотированным классам разрешено использовать класс `EntityManager`.

Проверка уровней

Инструменты управления, подобные ArchUnit, в том числе используются для того, чтобы обеспечить соблюдение принятых архитектурных решений. Архитекторы часто внедряют такие решения, как разделение ответственности, которые в краткосрочной перспективе вызывают неудобства для разработчиков, но с точки зрения эволюции и изоляции имеют долгосрочные преимущества. Рассмотрим рис. 4.14.

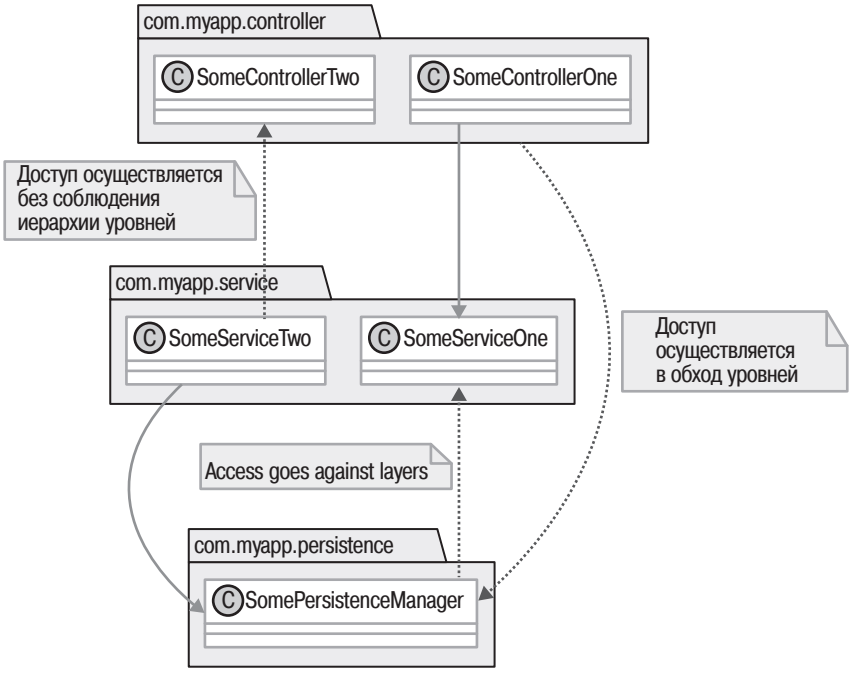


Рис. 4.14. Использование компонентов для определения многоуровневой архитектуры

Архитектор построил многоуровневую структуру, чтобы изолировать изменения между слоями (уровнями). В такой архитектуре зависимости должны существовать только между соседними слоями; чем больше слоев связываются с определенным слоем, тем больше побочных эффектов несут за собой изменения.

Код ArchUnit для фитнес-функции проверки уровней см. в примере 4.8.

Пример 4.8. Проверки управления многоуровневой архитектурой

```
layeredArchitecture()
    .consideringAllDependencies()
    .layer("Controller").definedBy("..controller...")
    .layer("Service").definedBy("...service...")
    .layer("Persistence").definedBy("..persistence...")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

В примере 4.8 архитектор определяет уровни и правила доступа для них.

Наверняка многие из вас, будучи архитекторами, описывали принципы из предыдущих примеров где-нибудь в вики или других хранилищах общедоступной информации, но ваши записи никто не читал! Декларировать принципы здорово, но если они не исполняются, то это скорее просто пожелания, а не принципы управления. Многоуровневая архитектура из примера 4.8 отлично это иллюстрирует — хотя архитектор может составить документ с описанием уровней и базового принципа разделения ответственности, он никогда не сможет быть уверен, что разработчики будут придерживаться этого документа, если не введет фитнес-функцию для проверки.

Мы много времени уделили разбору инструмента ArchUnit, поскольку это наиболее зрелый из многих ориентированных на управление фреймворков тестирования. Очевидно, что он применим только в экосистеме Java. К счастью, его стиль повторяет NetArchTest (https://github.com/BenMorris/NetArchTest), имеющий все основные возможности ArchUnit, но для платформы .NET.

Линтеры для управления кодом

Архитекторы, не работающие на Java и .NET, часто спрашивают, существует ли инструмент для платформы *X*, аналогичный ArchUnit. Хотя такие специфические инструменты, как ArchUnit, встречаются редко, для большинства

языков программирования существуют *линтеры* — утилиты, которые сканируют исходный код на предмет антипаттернов и различных недостатков. Как правило, линтер проводит лексический анализ и парсинг исходного кода и предоставляет разработчикам плагины для проверок синтаксиса. Например, в ESLint (https://eslint.org/), линтере для JavaScript (технически линтере для ECMAScript), можно прописать требование (или отсутствие требования) точки с запятой, необязательность скобок и так далее. Кроме того, в линтере можно задать правило, устанавливающее применимые политики вызова функций, и другие правила управления.

Линтеры есть на большинстве платформ; например, для C++ применяется Cpplint (https://github.com/cpplint/cpplint), а для языка Go — Staticcheck (https://staticcheck.io/). Существуют даже линтеры для SQL, включая sql-lint (https://github.com/joereynolds/sql-lint). Хотя они не так удобны, как ArchUnit, с их помощью можно добавить разнообразные структурные проверки практически в любую кодовую базу.

Практический пример: фитнес-функция доступности

Вопрос, волнующий многих архитекторов: использовать ли унаследованную систему (legacy system) в качестве точки интеграции или строить новую? Если разрабатываемый инструмент ранее еще не применялся, как в этом случае принять объективное решение?

PenultimateWidgets столкнулась с этой проблемой при интеграции с унаследованной системой. Для ее решения команда создала фитнес-функцию для стресс-тестирования унаследованного сервиса, как показано на рис. 4.15.

После настройки экосистемы команда измерила процент ошибок по сравнению с общим количеством ответов от сторонней системы с помощью инструмента мониторинга.

Результаты эксперимента показали, что унаследованная система не имела проблем с доступностью, хотя при этом и возникали большие накладные расходы для обработки точки интеграции.

Получив этот объективный результат, команда поняла, что унаследованной точки интеграции достаточно и можно высвободить ресурсы, которые в противном случае ушли бы на переписывание системы. Этот пример иллюстрирует, как благодаря фитнес-функциям разработка становится не интуитивной, а измеримой инженерной дисциплиной.

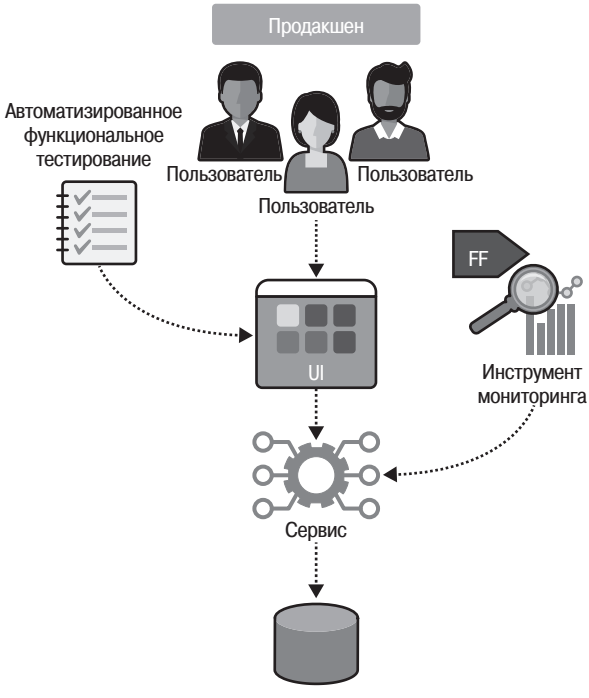


Рис. 4.15. Фитнес-функция для проверки доступности

Практический пример: нагрузочное тестирование и канареечные релизы

Служба PenultimateWidgets размещается на одной виртуальной машине. Однако под нагрузкой этот единственный экземпляр с трудом обеспечивает необходимую масштабируемость. В качестве быстрого временного решения команда внедряет автомасштабирование службы, реплицируя единственный экземпляр на несколько экземпляров, поскольку у бизнеса скоро начнется ежегодная распродажа. Однако скептически настроенные члены команды хотят быть уверенными, что новая система будет работать под нагрузкой.

Архитекторы проекта создали фитнес-функцию, привязанную к флагу функции (feature flag), которая позволяет осуществлять *канареечные релизы* (canary release) или *темные запуски* (dark launch), когда новая функциональность выпускается для ограниченного количества пользователей, чтобы проверить потенциальное влияние, вызванное новым изменением. Например, если раз-

работчики высокомасштабируемого сайта внедряют новую функцию, требующую высокой пропускной способности, они обычно предпочитают выпускать изменение постепенно, чтобы проследить его влияние. Этот пример показан на рис. 4.16.

В примере на рис. 4.16 команда сначала выпустила автомасштабируемые экземпляры для небольшой группы, а затем увеличила количество пользователей, поскольку мониторинг показал стабильную хорошую производительность и поддержку.

Это решение будет играть роль строительных лесов (скаффолдинга) для ограниченного расширения, пока команда разрабатывает лучшее решение. Имея регулярно выполняющуюся фитнес-функцию, команда сможет точнее оценить, в течение какого времени временное решение останется эффективным.

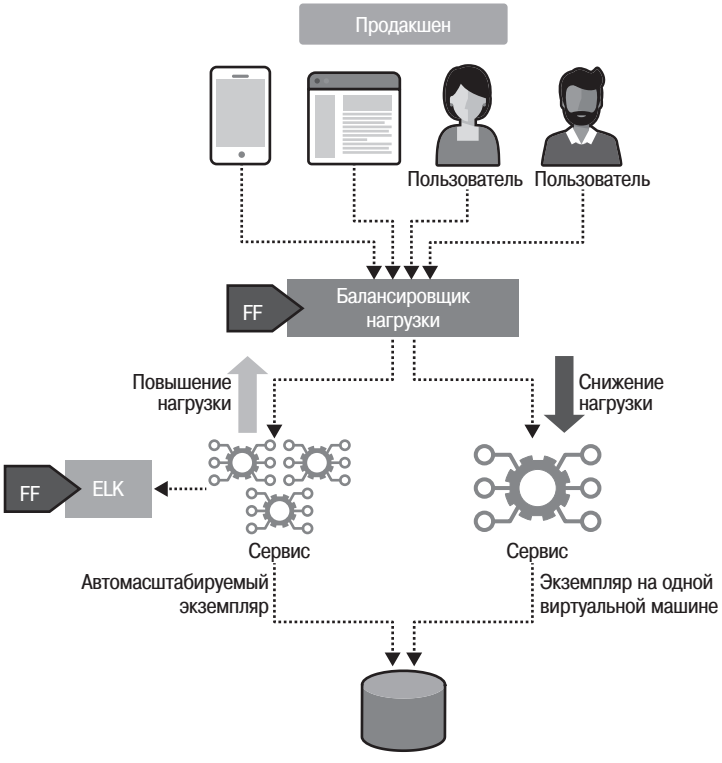


Рис. 4.16. Автомасштабирование канареечного релиза для обеспечения поддержки и повышения уверенности

Практический пример: что переносить

Приложение PenultimateWidgets, разработанное с использованием Java Swing почти десять лет назад и постоянно обрастающее новыми функциями, до сих пор было единственной «рабочей лошадкой» компании. Было решено перенести его в веб-среду. Однако перед бизнес-аналитиками встал сложный вопрос: какую часть существующей разросшейся функциональности переносить? И второй, более практический: в каком порядке внедрять перенесенные функции нового приложения, чтобы максимально быстро обеспечить наибольшую функциональность?

Один из архитекторов PenultimateWidgets спросил бизнес-аналитиков, какие функции являются самыми популярными, и оказалось, что они не имели об этом ни малейшего представления! Несмотря на то что они не один год уточняли детали приложения, у них не было реального понимания того, как пользователи взаимодействуют с ним. Чтобы получить информацию от пользователей, разработчики выпустили новую версию унаследованного приложения, отслеживавшую, какие функции меню были наиболее востребованными.

Через несколько недель аналитики собрали результаты и составили наглядную дорожную карту (road map) того, какие функции и в каком порядке переносить. Они обнаружили, что чаще всего используются функции выставления счетов и поиска информации о клиентах. Удивительно, но один из разделов, создание которого отняло множество усилий, использовался очень редко, и в результате команда решила убрать эту функциональность из нового веб-приложения.

Фитнес-функции, которые вы уже используете

За исключением таких инструментов, как ArchUnit, многие из описанных здесь решений и подходов не новы. Однако команды используют их редко, от случая к случаю. Частью понимания концепции *фитнес-функций* является то, что они объединяют в единую перспективу широкий спектр инструментов. Таким образом, скорее всего, вы уже используете фитнес-функции в своих проектах, просто пока не называете их так.

Фитнес-функции включают в себя наборы метрик, такие как SonarCube; линтеры, такие как esLint, pyLint и cppLint; и целое семейство инструментов проверки исходного кода, такое как PMD.

Использование мониторов для наблюдения за трафиком не делает их фитнес-функциями. Фитнес-функцией они становятся при наличии *объективной величины* с привязкой к оповещению.

Чтобы преобразовать метрику или измерение в фитнес-функцию, необходимо определить ее *объективную величину* и обеспечить быструю обратную связь, чтобы использовать надлежащим образом.

Если инструменты используются нерегулярно, их нельзя назвать фитнес-функциями; фитнес-функцией считается инструмент, выполняющий проверки на постоянной основе.

Интеграционная архитектура

Хотя многие фитнес-функции применяются к отдельным приложениям, они способны работать во всех частях экосистемы, которые можно улучшить с помощью управления. Естественно, что чем меньше задач будут использовать общие свойства приложения, тем меньше для них будет общих решений. Интеграционная архитектура по своей природе подразумевает объединение специфических частей, что не позволяет давать универсальные рекомендации. Тем не менее для фитнес-функций интеграционной архитектуры существуют некоторые общие паттерны.

Управление взаимодействием в микросервисах

Многие архитекторы стремятся внедрять для распределенных архитектур типа микросервисов тесты на циклы, как на рис. 2.3. Однако этому препятствует неоднородная природа архитектурных проблем. Тестирование на циклы компонентов — это проверка, проводимая во время компиляции, она требует единой кодовой базы и инструмента на соответствующем языке. Однако в микросервисах одного инструмента недостаточно: каждый сервис может быть написан в разных технологических стеках, в разных репозиториях, использовать разные протоколы связи и многие другие переменные. Таким образом, найти готовый инструмент для фитнес-функций микросервисов едва ли возможно.

Архитекторам часто приходится писать собственные фитнес-функции, но для этого не обязательно создавать целый фреймворк (к тому же это слишком трудоемкий процесс). Обычная фитнес-функция — это 10–15 строк «склеенного» кода, часто в технологическом стеке, отличающемся от того, в каком написано само решение.

Рассмотрим проблему управления вызовами между микросервисами, показанную на рис. 4.17. По замыслу архитектора, `OrderOrchestrator` – единственный владелец состояния рабочего процесса. Но если сервисы предметной области взаимодействуют друг с другом, оркестратор не может поддерживать корректное

состояние. Таким образом, необходимо обеспечить управление взаимодействием между сервисами: сервисы предметной области должны взаимодействовать только с оркестратором.

Однако если архитектор способен создать согласованный интерфейс для систем (например, ведения журналов в анализируемом формате), он может написать несколько строк кода на языке сценариев для создания фитнес-функции управления. Рассмотрим сообщение журнала (логирование), содержащее следующие данные:

- Название сервиса
- Имя пользователя
- IP-адрес
- Идентификатор корреляции
- Время получения сообщения по UTC
- Затраченное время
- Название метода

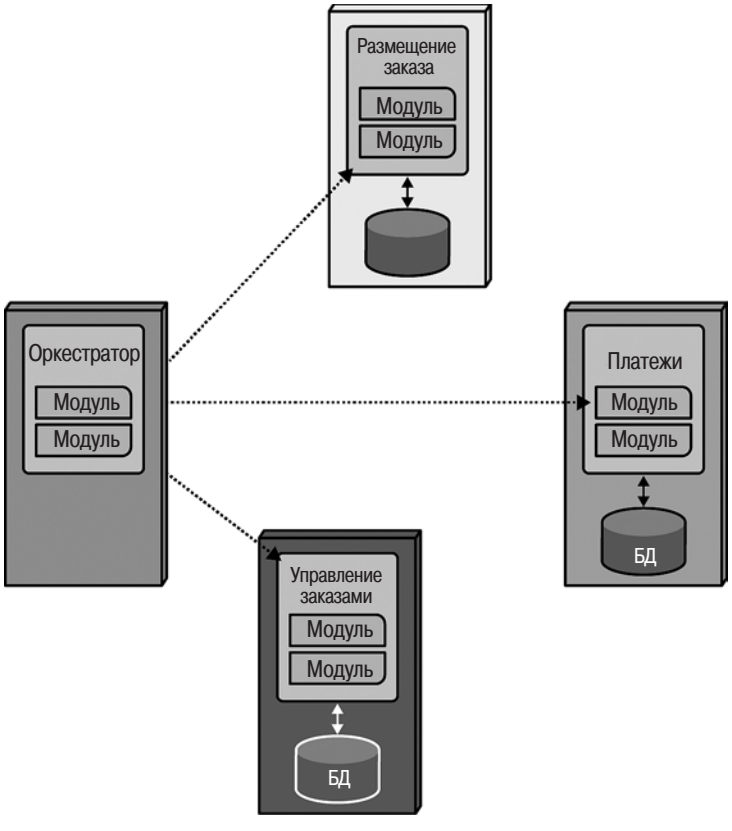


Рис. 4.17. Управление взаимодействием между микросервисами

Сообщение журнала может выглядеть как в примере 4.9.

Пример 4.9. Пример формата журнала микросервисов

```
["OrderOrchestrator", "jdoe", "192.16.100.10", "ABC123",
    "2021-11-05T08:15:30-05:00", "3100ms", "updateOrderState()"]
```

Во-первых, архитектор может создать фитнес-функцию для каждого проекта, которая будет проверять, что сообщения журнала выводятся в формате, показанном в примере 4.9, независимо от технологического стека. Эта фитнес-функция может быть прикреплена к образу контейнера, совместно используемого сервисами.

Во-вторых, архитектор пишет простую фитнес-функцию на языке сценариев, таком как Ruby или Python, собирающую сообщения журнала, проводящую парсинг общего формата, заданного в примере 4.9, и проверяющую допустимость взаимодействия, как показано в примере 4.10.

Пример 4.10. Проверка взаимодействия между сервисами

```
list_of_services.each { |service|
       service.import_logsFor(24.hours)
     calls_from(service).each { |call|
         unless call.destination.equals("orchestrator")
          raise FitnessFunctionFailure.new()
    }
  }
```

В примере 4.10 создан цикл, который итеративно просматривает все файлы журналов, собранные за последние 24 часа. Для каждой записи журнала проверяется, что местом назначения любого вызова является оркестратор, а не другой сервис предметной области. Если какой-то сервис нарушает это правило, фитнес-функция выдает исключение.

Возможно, вы узнали части этого примера из главы 2, где обсуждались *триггерные* и *непрерывные* фитнес-функции; это хороший пример двух способов реализации фитнес-функции с учетом различных компромиссов. В примере 4.10 показана *реактивная* фитнес-функция — она выполняет проверку управления через определенный промежуток времени (в данном случае 24 часа). Второй способ реализации этой фитнес-функции — *проактивный*, основанный на мониторинге взаимодействия в реальном времени и выявляющий нарушения по мере их возникновения.

Каждый подход подразумевает компромиссы. Реактивная версия во время выполнения не создает накладных расходов для характеристик архитектуры, в то время как мониторы могут добавить небольшие накладные расходы. Однако проактивная версия выявляет нарушения сразу, а не через день.

Таким образом, реальный компромисс между этими двумя подходами сводится к важности управления. Например, если несанкционированное взаимодействие немедленно вызывает сбой (например, сбой безопасности), фитнес-функция должна быть реализована проактивно. Если же ее цель — управление структурой, то реактивная фитнес-функция на основе журнала окажет меньше влияния на работу системы.

Практический пример: выбор способа реализации фитнес-функции

Тестирование предметной области обычно устроено просто: разработчики тестируют функции инкрементно по мере их реализации в коде, используя один или несколько фреймворков тестирования. Однако даже простые фитнес-функции имеют множество вариантов реализации.

Рассмотрим пример на рис. 4.18.

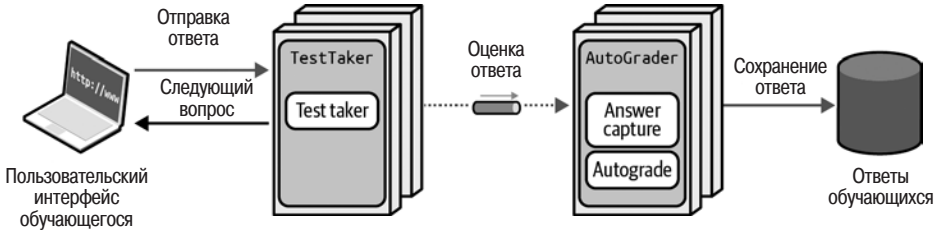


Рис. 4.18. Управление сообщениями для выставления оценок

На рис. 4.18 студент отвечает на вопросы теста, отправляемые ему сервисом `TestTaker`, который, в свою очередь, асинхронно передает сообщения в `AutoGrader`, сохраняющий ответы с оценкой. Ключевое требование для этой системы — надежность: ни один ответ не должен быть потерян при передаче. Как разработать фитнес-функцию для этой задачи?

Существуют как минимум два решения, различающиеся в основном компромиссами. Рассмотрим решение, показанное на рис. 4.19.

В современной архитектуре микросервисов такие свойства, как порты сообщений, обычно управляются в контейнере. Простой способ реализовать фитнес-функцию, показанную на рис. 4.19, — использовать контейнер для проверки количества входящих и исходящих сообщений и подавать оповещение, если эти значения не совпадают.

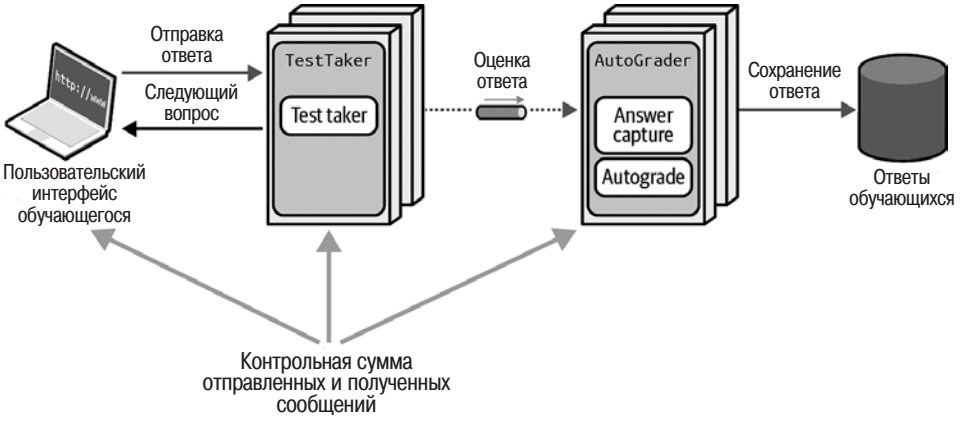


Рис. 4.19. Подсчет количества отправленных и полученных сообщений

Это простая фитнес-функция, поскольку она атомарна на уровне сервиса/контейнера, и архитекторы могут обеспечить ее выполнение в согласованной инфраструктуре. Однако она гарантирует надежность только на уровне отдельного сервиса, но не всей системы.

Альтернативный способ реализации фитнес-функции представлен на рис. 4.20.

На рис. 4.20 архитектор использует *идентификаторы корреляции* — распространенный метод, когда каждому запросу присваивается уникальный идентификатор для отслеживания. Чтобы обеспечить надежность передачи сообщения, при отправке каждому из них присваивается идентификатор корреляции и в конце процесса проверяется разрешение каждого идентификатора. Второй способ обеспечивает большую надежность передачи сообщений, но теперь придется поддерживать состояние всего рабочего процесса, что усложняет координацию.

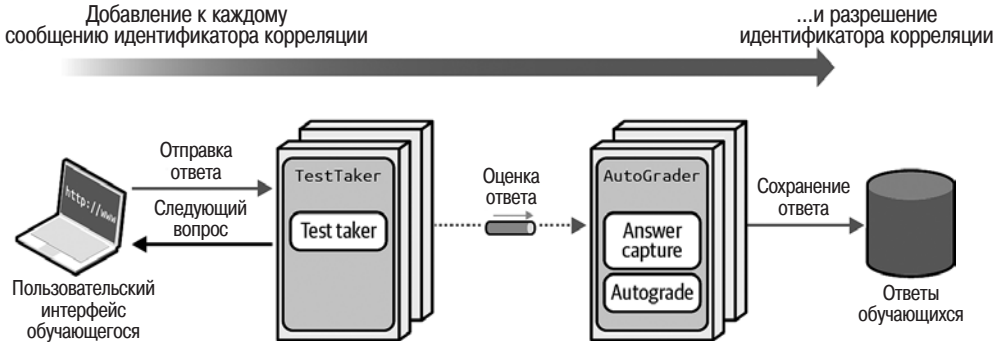


Рис. 4.20. Использование идентификаторов корреляции для обеспечения надежности

Какая реализация фитнес-функции является подходящей? Как и все в программной архитектуре, *это зависит от обстоятельств*! Выбор компромиссов часто обусловлен внешними обстоятельствами; главное — не зацикливаться на том, что существует только один способ реализации фитнес-функции.

Диаграмма на рис. 4.21 — пример реального проекта, в котором для обеспечения надежности данных была задана именно такая фитнес-функция.

Как видим, фитнес-функция выявила, что некоторые сообщения *не* проходили, что побудило команду провести ретроспективный анализ причин (и оставить фитнес-функцию работать, чтобы исключить возникновение проблем в будущем).

Ratio of "No. of non-missed exits event for the day"/"total number of exits for the day"

Trend Reports in Google Data studio#

	effectiveDa...	total_exits_per_day	successful_exits_per_day	service_level_indicator
1.	Mar 28, 2022	5	5	1
2.	Mar 27, 2022	4	4	1
3.	Mar 25, 2022	17	16	0,94
4.	Mar 24, 2022	1	1	1
5.	Mar 23, 2022	9	9	1
6.	Mar 22, 2022	4	4	1
7.	Mar 21, 2022	10	10	1
8.	Mar 20, 2022	1	1	1
9.	Mar 18, 2022	10	10	1
10.	Mar 17, 2022	24	24	1
11.	Mar 16, 2022	1	1	1
12.	Mar 15, 2022	6	6	1
13.	Mar 14, 2022	4	4	1
14.	Mar 13, 2022	1	1	1

1-25/25 < >

Рис. 4.21. Диаграмма, показывающая надежность сообщений в оркестрованном рабочем процессе (отношение «количество непропущенных событий выхода за день» / «общее количество событий выхода за день»)

DevOps

Хотя большинство фитнес-функций, которые мы рассматриваем, относятся к архитектурной структуре и связанным с ней концепциям, таким как собственно программная архитектура, проблемы управления могут затрагивать все части экосистемы, включая фитнес-функции, относящиеся к DevOps.

Это именно фитнес-функции, а не просто эксплуатационные характеристики, по двум причинам. Во-первых, они служат точкой пересечения архитектуры и эксплуатации — изменения в архитектуре могут повлиять на особенности эксплуатации системы. Во-вторых, они заключают в себе проверки управления, дающие объективные результаты.

ХАОС-ИНЖИНИРИНГ

Когда инженеры разрабатывали распределенную архитектуру Netflix, они проектировали ее для работы в облаке Amazon Cloud. Но их беспокоили проблемы с высокой задержкой, доступностью, эластичностью и т. д., которые могли возникнуть из-за отсутствия прямого контроля над операциями. Тогда разработчики создали инструмент Chaos Monkey, «обезьяну хаоса», за которым последовало целое семейство инструментов Simian Army (армия обезьян) (https://github.com/Netflix/SimianArmy) с открытым исходным кодом. В то время как оригинальный Chaos Monkey предназначался для управления хаосом и случайностью, расширенный пакет Simian Army включал специализированные средства:

Chaos Monkey

Chaos Monkey проникает в центр обработки данных Amazon и инициирует непредвиденные события: увеличение задержки, снижение надежности и другой хаос. Проектируя с Chaos Monkey, каждая команда должна создавать устойчивые сервисы, способные противостоять навязанному хаосу.

Chaos Gorilla

Chaos Gorilla может вывести из строя целый центр обработки данных Amazon, внезапно симулируя полное отключение данных.

Chaos Kong

Если Chaos Gorilla окажется недостаточно устрашающим, то Chaos Kong выведет из строя целую зону доступности, в результате чего часть облачной экосистемы просто исчезнет. Словно в подтверждение эффективности Chaos Engineering в целом и Simian Army в частности, несколько лет назад (https://oreil.ly/2pv4V) недостаточная автоматизация привела к тому, что инженер Amazon случайно отключил весь Amazon East (он перепутал команду и вместо

`kill 10` задал `kill 100`). Однако во время этого сбоя Netflix оставался работоспособным — благодаря Chaos Gorilla архитекторы при написании кода предусмотрели обход этой случайности.

Doctor Monkey

Doctor Monkey может проверять общее состояние службы — использование процессора, дискового пространства и т. д. — и выдавать оповещения, если выявляет ограничения для какого-либо ресурса.

Latency Monkey

Одной из постоянных проблем облачных ресурсов, особенно на первых порах, была высокая задержка при получении данных. Хотя оригинальный Chaos Monkey также мог случайным образом затрагивать задержку, стрессор Latency Monkey был создан специально для тестирования этого параметра.

Janitor Monkey

Экосистема Netflix эволюционирует: появляются новые сервисы, которые постепенно заменяют и улучшают существующие, но поточность изменений не предполагает, что нужно обязательно переходить на новые возможности сразу после их появления. Наоборот, их можно использовать, когда это становится удобно. Отсутствие формального цикла выпуска у многих сервисов ведет к появлению *осиротевших* сервисов — таких, которые все еще работают в облаке, но их никто не использует, поскольку все пользователи перешли на более новую версию. Janitor Monkey (обезьяна-уборщик) решает эту проблему путем поиска сервисов, все еще работающих в облаке, но не получающих вызовов от других сервисов, и удаляет этих «сирот» из облака, экономя при этом взаимозаменяемые облачные ресурсы, которые они потребляли.

Conformity Monkey

Conformity Monkey предоставляет архитекторам Netflix платформу для реализации конкретных фитнес-функций управления. Например, архитекторам может быть важно, чтобы все конечные точки REST поддерживали допустимые команды, правильно обрабатывали ошибки и поддерживали метаданные должным образом, и поэтому они создадут инструмент, который будет постоянно вызывать конечные точки REST (так же, как это делают обычные клиенты) для проверки результатов.

Security Monkey

Как следует из названия, Security Monkey — это версия Conformity Monkey, специализирующаяся на вопросах безопасности. Например, она выполняет сканирование в поисках открытых портов отладки, отсутствия процедуры аутентификации и другие автоматизированные проверки.

Simian Army имел открытый исходный код и в конечном итоге перестал поддерживаться, поскольку инженеры Netflix создали более совершенные механизмы

управления. Однако некоторые из обезьянок обрели новый дом. Например, очень полезная Janitor Monkey возродилась в Swabbie (https://github.com/spinnaker/swabbie) как часть набора облачных фитнес-функций с открытым исходным кодом.

В основе Chaos Engineering лежит убедительный принцип: вопрос не в том, *произойдет ли* в системе сбой, а в том, *когда* он произойдет. Моделируя управляемые случайности, архитекторы и группа эксплуатации могут вместе создавать более надежные системы.

Необходимо отметить, что Chaos Monkey не запускался по графику, а работал непрерывно в экосистеме Netflix. Такой режим позволял разработчикам не только создавать устойчивые системы, но и постоянно проверять их работоспособность. Благодаря встроенным в архитектуру постоянным проверкам Netflix создал одну из самых надежных систем в мире. Simian Army — отличный пример целостной, непрерывной, эксплуатационной фитнес-функции. Она проверяла сохранение характеристик архитектуры (отказоустойчивости, масштабируемости и т. д.) одновременно в нескольких ее частях.

Архитектура предприятия

Большинство фитнес-функций, которые мы рассматривали до сих пор, касались архитектуры приложений или интеграции, но они применимы к любой архитектуре, которую управление может улучшить. В частности, архитекторы систем предприятий оказывают большое влияние на остальную экосистему, когда определяют *платформы* внутри экосистемы, инкапсулирующие бизнес-функциональность. Это согласуется с нашим стремлением сохранить детали реализации в минимально возможном объеме.

Рассмотрим пример на рис. 4.22.

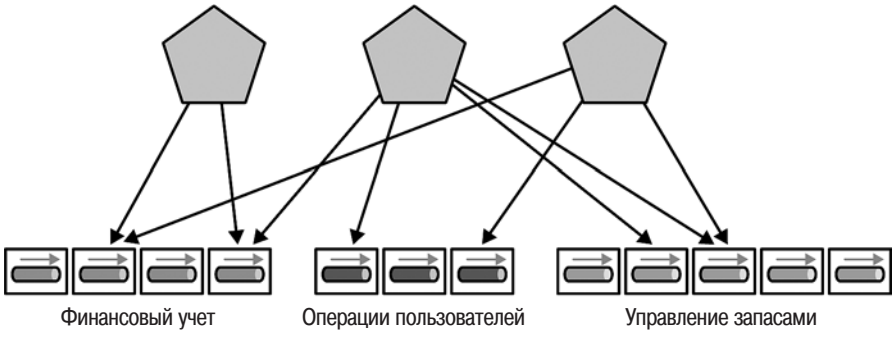


Рис. 4.22. Приложения как специализированные наборы сервисов

На рис. 4.22 приложения (вверху) потребляют сервисы из разных частей системы предприятия. Точечный доступ приложений к сервисам приводит к тому, что детали реализации их взаимодействия просачиваются в приложения, которые становятся более уязвимыми.

Понимая это, многие архитекторы систем предприятий разрабатывают платформы для инкапсуляции бизнес-функциональности за управляемыми контрактами, как показано на рис. 4.23.

На рис. 4.23 мы видим платформы, скрывающие рабочие взаимодействия за последовательным, медленно изменяющимся API, который описывает средства, необходимые другим частям экосистемы, используя контракты для платформы. Инкапсулируя детали реализации на уровне платформы, архитекторы препятствуют расширению связанности в реализации, что, в свою очередь, делает архитектуру более надежной.

Архитекторы системы предприятия определяют API для этих функций платформы и фитнес-функции, управляющие возможностями, структурой и другими характеристиками платформы и ее реализацией. Это позволяет им *не заниматься* выбором технологий и сосредоточиться на возможностях, а не на том, как их реализовать, что решает сразу две задачи.

Прежде всего, архитекторы предприятий, как правило, далеки от деталей реализации и поэтому не в курсе самых последних изменений как в технологиях в целом, так и внутри собственной экосистемы; они часто попадают под действие антипаттерна «Замороженный троглодит» (Frozen Caveman).

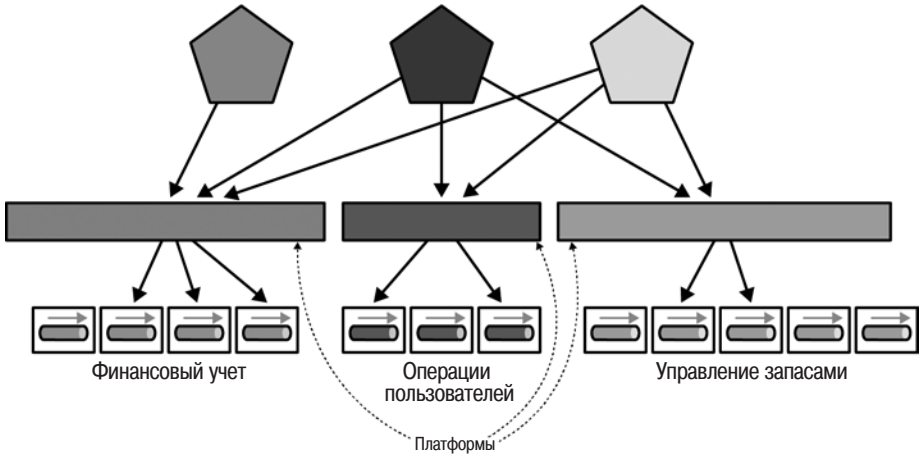


Рис. 4.23. Построение платформ для скрытия деталей реализации

АНТИПАТТЕРН «ЗАМОРОЖЕННЫЙ ТРОГЛОДИТ»

Часто встречающийся поведенческий антипаттерн *Замороженный троглодит* (Frozen Caveman) описывает архитектора, испытывающего одно и то же иррациональное беспокойство по поводу любой архитектуры. Например, один из коллег Нила работал над системой с централизованной архитектурой. Но каждый раз, когда он представлял проект архитекторам заказчика, они задавали вопрос: «А мы не потеряем Италию?» За несколько лет до этого из-за странной проблемы с коммуникацией головной офис не мог связаться с магазинами в Италии, что привело к большим неудобствам. Хотя вероятность повторения этой ситуации была крайне мала, она не давала покоя архитекторам.

Как правило, этот антипаттерн проявляется у тех, кто в прошлом обжегся на неудачном решении или столкнулся с неожиданным инцидентом и теперь дует на воду. Хотя оценка рисков важна, она должна быть реалистичной. Архитекторы должны учиться определять разницу между реальным и мнимым техническим риском. Чтобы мыслить как архитектор, необходимо отойти от точки зрения «замороженного троглодита», видеть другие решения и ставить более актуальные вопросы.

Какие бы устаревшие тенденции внедрения ни использовали архитекторы систем предприятий, они лучше всего понимают долгосрочные стратегические цели организации, которые могут передать в фитнес-функциях. Вместо того чтобы заниматься выбором технологий, они смогут задавать соответствующие фитнес-функции на уровне платформы, чтобы та продолжала поддерживать необходимые характеристики и поведение. Поэтому мы также советуем раскладывать характеристики архитектуры на составные части до тех пор, пока они не станут объективно измеримыми — а тем, что можно измерить, можно управлять.

Кроме того, если позволить архитекторам системы предприятия сосредоточиться на построении фитнес-функций, управляющих стратегическим видением, архитекторы предметной области и интеграции смогут принимать технологические решения, последствия которых будут защищены барьером фитнес-функций. Это, в свою очередь, позволит организациям вырастить новое поколение архитекторов, поскольку специалисты более низкого уровня смогут принимать решения и работать над компромиссами.

Мы консультировали компании, в которых роль архитектора системы предприятия является *эволюционной*. Этот специалист ответствен за поиск и реализацию фитнес-функций (часто расширяемых из реализации конкретного проекта) и создание многократно используемых экосистем с соответствующими квантовыми границами и контрактами для обеспечения свободной связи между платформами.

Практический пример: изменение архитектуры при развертывании 60 раз в день

GitHub (http://github.com/) — известный ресурс для разработчиков — деплоит (выпускает развертывания) в среднем 60 раз в день. В своем блоге Move Fast and Fix Things (https://github.blog/2015-12-15-move-fast/) GitHub описывает проблему, от которой у многих архитекторов мурашки побегут по коже. Оказывается, GitHub уже давно использовал для обработки слияний сценарий оболочки командной строки Git, который работает правильно, но недостаточно хорошо масштабируется. Команда разработчиков Git создала библиотеку libgit2 для замены многих функций командной строки Git и реализовала в ней функциональность слияния, проведя ее тщательное локальное тестирование.

Но далее было необходимо развернуть новое решение на продакшен. Существующая функциональность работала безупречно с самого момента создания GitHub. Разработчики очень боялись ее нарушить, но им также было необходимо решить проблему технического долга.

К счастью, разработчики GitHub создали Scientist (https://github.com/github/scientist), фреймворк с открытым исходным кодом, написанный на Ruby, в котором можно проводить комплексное, непрерывное тестирование изменений в коде. В примере 4.11 приведена структура теста `Scientist`.

Пример 4.11. Тестовые настройки Scientist

```
require "scientist"

class MyWidget
  include Scientist

  def allows?(user)
    science "widget-permissions" do |e|
      e.use { model.check_user(user).valid? } # прежний способ
      e.try { user.can?(:read, model) } # новый способ
    end # возвращает контрольное значение
  end
end
```

В примере 4.11 существующее поведение инкапсулируется в блоке `use` (называемом *контрольным*), а экспериментальное поведение добавляется в блок `try` (называемый *кандидатом*). Блок `science` обрабатывает следующие задачи во время вызова кода:

Решает, нужно ли запускать блок `try`

Настройки Scientist определяют, как будет проходить эксперимент. В данном примере, целью которого было обновление функциональности слия-

ния, 1 % случайных пользователей применяли новую функциональность. В любом случае Scientist *всегда* возвращает результаты блока `use`, поэтому в случае различия результатов вызывающая сторона всегда будет получать существующее поведение.

Определяет случайный порядок выполнения блоков `use` *и* `try`

Scientist выполняет эту задачу, чтобы избежать случайной маскировки ошибок из-за неизвестных зависимостей. Иногда порядок или другие случайные факторы могут вызывать ложные срабатывания; после рандомизации вероятность этих ошибок снижается.

Измеряет продолжительность всех сценариев поведения

Scientist, кроме прочего, выполняет A/B тестирование производительности, поэтому в него встроен мониторинг производительности. Разработчики могут использовать фреймворк по частям — например, для тестирования вызовов без их моделирования.

Сравнивает результат выполнения `try` *с результатом выполнения* `use`

Поскольку целью Scientist является рефакторинг существующего поведения, инструмент сравнивает и регистрирует результаты каждого вызова, чтобы определить, существуют ли различия.

Скрывает (но регистрирует) любые исключения, возникающие в блоке `try`

Всегда существует вероятность, что новый код будет выбрасывать непредвиденные исключения. Они не должны быть заметны конечным пользователям, поэтому инструмент делает их невидимыми для них (но регистрирует их для дальнейшего анализа разработчиками).

Публикует все свои данные

Инструмент Scientist предоставляет все свои данные в различных форматах.

Для тестирования новой реализации слияния разработчики GitHub использовали следующий вызов (под названием `create_merge_commit_rugged`), как показано в примере 4.12.

В примере 4.12 вызов `create_merge_commit_rugged` произошел в 1 % обращений, но в масштабе GitHub все крайние случаи проявляются быстро.

При выполнении этого кода результат для конечных пользователей всегда корректный. Если блок `try` возвращает значение, отличное от `use`, это фиксируется и возвращается значение `use`. Таким образом, в худшем случае конечные пользователи получат результат, который они получали до рефакторинга. После тестирования в течение 4 дней и отсутствия случаев замедления работы или несовпадения результатов в течение 24 часов подряд разработчики удалили старый код слияния и оставили новый.

Пример 4.12. Тестирование нового алгоритма слияния

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Merge #{head} into #{base}"
  now = Time.current

  science "create_merge_commit" do |e|
    e.context :base => base.to_s, :head => head.to_s, :repo => repository.nwo
    e.use { create_merge_commit_git(author, now, base, head, commit_message) }
    e.try { create_merge_commit_rugged(author, now, base, head, commit_message)
}
  end
end
```

С нашей точки зрения, Scientist — это фитнес-функция. Это отличный практический пример стратегического использования комплексной, непрерывной фитнес-функции, обеспечившей уверенное проведение рефакторинга критически важной части инфраструктуры. Разработчики изменили ключевую часть архитектуры, запустив новую версию вместе с существующей, по сути, превратив унаследованную реализацию в тестирование их согласованности.

Фитнес-функции соответствия

Инструмент Scientist реализует общий тип проверки, называемый *фитнес-функцией соответствия* (fidelity fitness function): сохранение соответствия между новой системой и старой, подлежащей замене. Многие организации долгое время создают все новые важные функции и не проводят их достаточного тестирования, пока не приходит время заменить технологию, но при этом сохранить поведение старой системы. Чем старше система и чем хуже она документирована, тем сложнее разработчикам воспроизвести желаемое поведение.

Фитнес-функция соответствия позволяет сравнивать части *старых* и *новых систем*. В процессе замены обе системы работают параллельно, и команда с помощью прокси вызывает старую, новую или обе системы контролируемым образом, пока не перенесет каждый бит дискретной функциональности. Некоторые команды неохотно применяют такой механизм, поскольку представляют сложность разделения старого поведения и точной репликации. Но в конечном итоге они уступают из-за необходимости достичь уверенности в результате.

Фитнес-функции — это инструмент проверки, а не принуждения

Мы понимаем, что вооружаем архитекторов своего рода палкой, которой они могут тыкать разработчиков; но наша цель вовсе не в этом. Архитекторы не должны замыкаться в башне из слоновой кости и придумывать все более сложные

фитнес-функции с большим количеством взаимосвязей, увеличивая нагрузку на разработчиков, но не добавляя ценности проекту.

Напротив, фитнес-функции — это способ обеспечить соблюдение принципов архитектуры. Во многих сферах, например в авиации и хирургии, используются (иногда в обязательном порядке) чек-листы. Они нужны не потому, что пилоты или хирурги не знают свою работу или забывчивы — чек-листы помогают избежать естественного риска случайно пропустить какой-то шаг (риска, которому подвержены люди, постоянно выполняющие сложные задачи). Например, каждый разработчик знает, что не следует развертывать контейнер с включенными портами отладки, но в режиме многозадачности он может забыть об этом.

Многие архитекторы описывают принципы архитектуры и дизайна системы в вики или других общедоступных базах знаний, но эти принципы отходят на второй план в условиях сжатых сроков и других ограничений. Выразив их в коде фитнес-функций, вы гарантируете, что их не пропустят под воздействием внешних факторов.

Фитнес-функции необходимо писать совместно с разработчиками, которые должны уметь понимать их и исправлять при случайных сбоях. Хотя фитнес-функции добавляют накладные расходы, они предотвращают постепенную деградацию кодовой базы (*гниение бит*) и позволяют ей продолжать развиваться.

Документирование фитнес-функций

Тесты — хорошая документация, потому что всегда можно выполнить их и проверить написанное. Доверяй, но проверяй!

Существуют разные способы документирования фитнес-функции — в соответствии с общепринятым в организации. Некоторые архитекторы считают, что в самих фитнес-функциях уже задокументирован их замысел. Однако в тестах (какими бы поверхностными они ни были) нетехническим специалистам разбираться сложнее.

Многие архитекторы предпочитают вести документацию в реестре архитектурных решений (Architectural Decision Records, ADR) (https://adr.github.io/). Команды, использующие фитнес-функции, добавляют в ADR раздел, описывающий, как управлять изложенными проектными решениями.

Другой вариант — использовать фреймворк разработки через поведение (behavior-driven development, BDD), такой как Cucumber (https://cucumber.io/). Эти инструменты предназначены для сопоставления родного языка с проверочным кодом. Посмотрим на тест Cucumber, приведенный в примере 4.13.

Пример 4.13. Предположения Cucumber

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told «Nope»
```

Функция `Feature`, описанная в примере 4.13, сопоставляется с методом языка программирования; сопоставление с Java показано в примере 4.14.

Пример 4.14. Методы Cucumber, сопоставляемые с описаниями

```
@Given("today is Sunday")
public void today_is_sunday() {
    // Написать код, который преобразует фразу выше в конкретное действие
    throw new io.cucumber.java.PendingException();
}
@When("I ask whether it's Friday yet")
public void i_ask_whether_it_s_friday_yet() {
    // Написать код, который преобразует фразу выше в конкретное действие
    throw new io.cucumber.java.PendingException();
}
@Then("I should be told {string}")
public void i_should_be_told(String string) {
    // Написать код, который преобразует фразу выше в конкретное действие
    throw new io.cucumber.java.PendingException();
}
```

Архитекторы могут использовать сопоставления между объявлениями на родном языке в примере 4.13 и определениями методов в примере 4.14, чтобы задавать фитнес-функции на более-менее простом родном языке и отображать их выполнение в соответствующем методе. Таким образом, они смогут документировать свои решения и одновременно выполнять их.

У использования такого инструмента, как Cucumber, есть один недостаток — несоответствие между фиксацией требований (изначальная задача инструмента) и документированием фитнес-функций.

В попытке объединить документацию и исходный код Дональд Кнут (Donald Knuth) ввел концепцию грамотного программирования (literate programming) с целью очистить документацию. Он создал специальные компиляторы для актуальных в то время языков, но не получил поддержки.

Однако в современных экосистемах такие инструменты, как Mathematica (https://www.wolfram.com/mathematica/) и записные книжки Jupyter (https://jupyter.org/), популярны, к примеру, в сфере data science. Записные книжки Jupyter, в частности, подходят для документирования и выполнения фитнес-функций.

В одном из примеров[1] команда создала записную книжку для проверки выполнения правил архитектуры с помощью структурного анализатора кода jQAssistant (https://jqassistant.org/) в сочетании с базой данных графов Neo4j (https://neo4j.com/). jQAssistant сканирует артефакты (байт-код Java, историю Git, зависимости Maven и т. д.) и сохраняет структурную информацию в базе данных Neo4j, как показано на рис. 4.24.

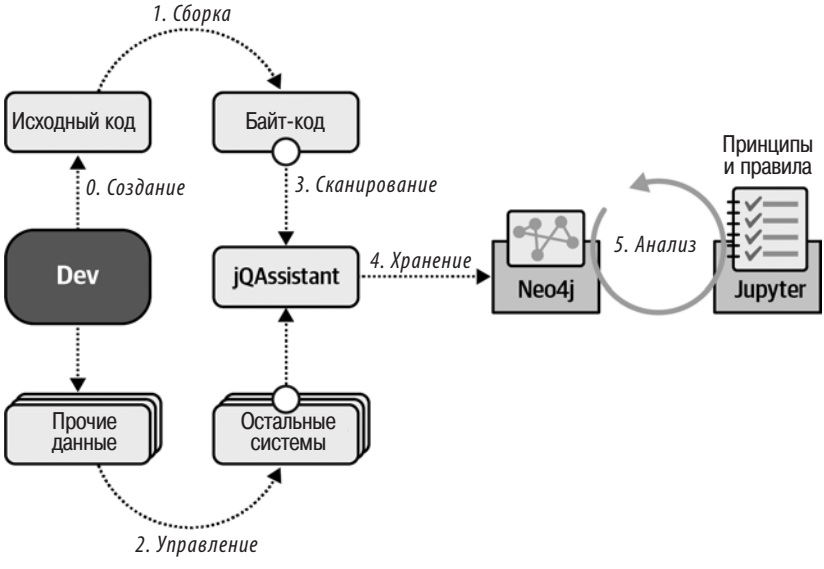


Рис. 4.24. Процесс управления с помощью записной книжки Jupyter

На рис. 4.24 связи между частями кодовой базы размещены в базе данных графов, что позволяет разработчикам выполнять запросы наподобие следующих:

```
MATCH (e:Entity)<-[:CONTAINS]-(p:Package)
WHERE p.name <> "model"
RETURN e.fqn as MisplacedEntity, p.name as WrongPackage
```

Вывод анализа, выполненного в приложении PetClinic, показан на рис. 4.25.

Out[4]:

MisplacedEntity	WrongPackage
org.springframework.samples.petclinic.repository.PetType	repository

Рис. 4.25. Результат анализа графов

[1] https://www.feststelltaste.de/checking-architecture-governance-with-jqassistant-neo4j-and-jupyter/

Результат, представленный на рис. 4.25, указывает на нарушение правил управления, когда все классы в пакете `model` должны реализовывать аннотацию `@Entity`.

С использованием записных книжек Jupyter архитекторы могут формулировать правила управления, а также выполнять их по требованию.

Важно документировать фитнес-функции, поскольку разработчики должны понимать их назначение и спокойно относиться к необходимости их исправления. Самый простой и последовательный способ добиться этого понимания — включить определения фитнес-функций в существующую корпоративную документацию. В первую очередь важно следить, чтобы фитнес-функции выполнялись, но нельзя забывать и о том, что они должны быть понятны тем, кто с ними работает.

Итоги

Фитнес-функции относятся к управлению архитектурой так же, как модульные тесты относятся к изменениям в предметной области. Однако реализация фитнес-функций зависит от факторов, составляющих конкретную архитектуру. Не существует универсальной архитектуры — каждая архитектура представляет собой уникальную комбинацию решений и технологий, на создание которой могли уйти годы или десятилетия. Это вынуждает архитекторов проявлять изобретательность при создании фитнес-функций. Однако это не требует написания целого фреймворка тестирования. Часто фитнес-функции пишутся на языке сценариев, таком как Python или Ruby, и представляют собой 10 или 20 строк «склеенного» кода, объединяющего выводы других инструментов. В примере 4.10 данные файлов журнала собираются и проверяются на наличие определенных строковых шаблонов.

На рис. 4.26 представлена отличная аналогия для фитнес-функций, которую придумал один из наших коллег.

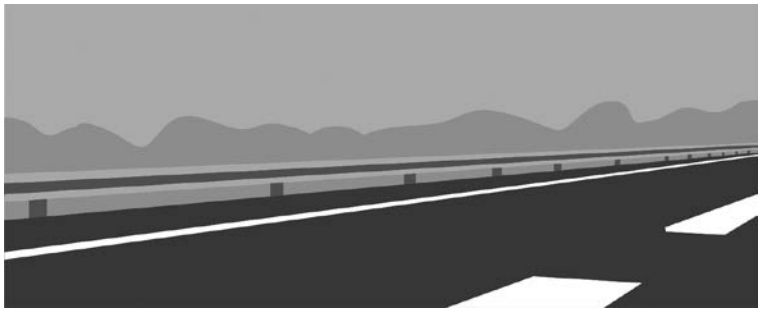


Рис. 4.26. Фитнес-функции выполняют роль отбойника независимо от материала дорожного покрытия

Материал дорожного покрытия (рис. 4.26) может быть совершенно разным — асфальт, булыжник, гравий и т. д. Отбойные ограждения предназначены для удержания автомобилей на полосе, независимо от того, какой это автомобиль и какая дорога. Фитнес-функции — это своего рода отбойники для характеристик архитектуры, создаваемые, чтобы предотвращать деградацию и поддерживать эволюцию систем с течением времени.

ЧАСТЬ II

Структура

Часть I определила *механику* эволюционной архитектуры — как команды разработчиков создают фитнес-функции, пайплайны развертывания и другие механизмы для управления программными проектами и их развития.

Часть II посвящена *структуре* архитектуры. Топология программной системы оказывает огромное влияние на ее способность к эволюции. Значительную часть работы архитектора составляет структурное проектирование, и грамотное соблюдение его принципов позволяют обеспечить более направленную эволюцию с течением времени.

В современных системах помимо собственно архитектуры большое влияние имеют данные и их эволюция, и мы также рассматриваем пересечение этих сфер.

ГЛАВА 5

Топологии эволюционной архитектуры

Дискуссии об архитектуре часто сводятся к связанности (coupling): каким образом части архитектуры соединяются и как они зависят друг от друга. Многие архитекторы рассматривают связанность как необходимое зло, поскольку невозможно построить сложный программный продукт, не связанный с другими компонентами. Для создания эволюционной архитектуры важно подобрать оптимальную связанность компонентов, обеспечивающую максимальную выгоду при минимальных накладных расходах и затратах.

Из этой главы читатели получат более глубокое представление о связанности архитектуры и о том, как она влияет на архитектурную структуру, и узнают, как оценить структуру программной архитектуры, чтобы обеспечить ее более эффективную эволюцию. Мы также разберем соответствующую терминологию и дадим несколько советов по топологии архитектуры от уровня компонентов до уровня всей системы.

Структура архитектуры, способной к эволюции

У разных стилей архитектуры разные эволюционные характеристики, но ни одна из них не определяет способность того или иного стиля к эволюции. Скорее эта способность зависит от характера связанности в архитектуре. Ключевой фактор эволюции ПО определяется по крайней мере в двух исследованиях, про которые рассказано ниже. Обе эти работы важны для понимания связанности в архитектуре.

Коннасценция

В 1996 году Мейлир Пейдж-Джонс (Meilir Page-Jones) опубликовал книгу «What Every Programmer Should Know About Object-Oriented Design» (Dorset House). В ней рассматривалась методика объектно-ориентированного проектирования, не снискавшая популярности. Однако ценность книги заключается в концепции, которую автор назвал *коннасценцией* (connascence). Этому понятию автор дал следующее определение:

> Два компонента являются коннасцентными, если изменения, внесенные в один компонент, потребуют модификации другого для сохранения корректной работы системы.
>
> — *Мейлир Пейдж-Джонс*

По сути, коннасценция — это усовершенствованное описание связанности. Оно отлично подходит для общения с техлидами и разработчиками, потому что с его помощью архитекторы могут более точно и выразительно описать, что представляет собой связанность и (что более важно) как ее улучшить. Более разнообразная терминология позволяет применить гипотезу Сепира — Уорфа.

ГИПОТЕЗА СЕПИРА — УОРФА

Принцип, утверждающий, что структура языка влияет на мировосприятие и когнитивные особенности его носителей, то есть что мышление людей определяется языком, на котором они говорят.

Например, в языках многих народов Крайнего Севера больше слов для обозначения *снега,* чем в языках людей, живущих на экваторе (которым не приходится регулярно различать виды снега). Можно утверждать, что жители Крайнего Севера имеют более глубокое представление о снеге.

Пейдж-Джонс определил два типа коннасценции: *статическую* и *динамическую*.

Статическая коннасценция

Статическая коннасценция описывает связанность на уровне исходного кода (в отличие от связанности во время выполнения, которое рассматривается в разделе «Динамическая коннасценция» на с. 117); она уточняет понятие центробежных (афферентных) и центростремительных (эфферентных) связей, определенное в книге «Structured Design». Другими словами, архитекторы рассматривают следующие типы статической коннасценции как *степень*, в ко-

торой что-то связывается либо по центробежной, либо по центростремительной траектории.

Коннасценция имени (Connascence of Name, CoN)

Связанные компоненты должны иметь единое имя сущности.

Имена методов — самый распространенный и предпочтительный способ связанности кодовых баз, особенно с учетом применения современных инструментов рефакторинга, которые значительно упрощают изменение имен в масштабах всей системы.

Коннасценция типа (Connascence of Type, CoT)

Связанные компоненты должны иметь единый тип сущности.

Этот тип коннасценции относится к распространенной во многих статически типизированных языках возможности ограничивать переменные и параметры определенными типами. Однако это не исключительно языковая особенность. Некоторые динамически типизированные языки предлагают выборочную типизацию, например Clojure (https://clojure.org/) и Clojure Spec (https://clojure.org/about/spec).

Коннасценция смысла (Connascence of Meaning, CoM), или коннасценция соглашения (Connascence of Convention, CoC)

Определенные параметры связанных компонентов должны быть одинаковы по смыслу.

Самым распространенным и очевидным примером этого типа коннасценции в кодовых базах являются жестко закодированные числа, а не константы. Например, в некоторых языках принято определять `int TRUE = 1; int FALSE = 0`. Представьте, что будет, если поменять эти значения местами.

Коннасценция позиции (Connascence of Position, CoP)

Порядок следования значений в связанных компонентах должен быть одинаковым.

Эта проблема со значениями параметров для вызовов методов и функций появляется даже в языках со статической типизацией. Например, если разработчик создает метод `void updateSeat(String name, String seatLocation)` и вызывает его со значениями `updateSeat("14D", "Ford, N")`, семантика будет неверной, даже если типы верны.

Коннасценция алгоритма (Connascence of Algorithm, CoA)

Связанные компоненты должны иметь единые алгоритмы.

Распространенный пример этого типа коннасценции — создание разработчиком безопасного алгоритма хеширования, который должен работать на

сервере и клиенте и выдавать одинаковые результаты для аутентификации пользователя. Очевидно, что для этого необходимо иметь сильную связанность: если в одном из алгоритмов что-то изменится, синхронизация будет нарушена.

Динамическая коннасценция

Другой тип коннасценции, определенный Пейджем-Джонсом, — это *динамическая коннасценция*, которая анализирует вызовы во время выполнения. Ниже описаны ее разновидности.

Коннасценция исполнения (Connascence of Execution, CoE)

Порядок выполнения связанных компонентов имеет значение.

Рассмотрим код:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

Он не будет работать, потому что свойства должны быть расположены по порядку.

Коннасценция синхронности (Connascence of Timing, CoT)

Время выполнения связанных компонентов имеет значение.

Распространенный пример данного типа коннасценции — состояние гонки, вызванное одновременным выполнением двух потоков, что влияет на результат совместной операции.

Коннасценция значений (Connascence of Values, CoV)

Взаимосвязанные значения должны изменяться вместе.

Рассмотрим пример прямоугольника, заданного четырьмя точками, представляющими его углы. Если требуется сохранить целостность структуры данных, не получится произвольно изменить одну из точек и при этом оставить другие точки неизменными.

Более частый и сложный пример связан с транзакциями, особенно в распределенных системах. Если в системе существуют отдельные базы данных и необходимо обновить одно конкретное значение во всех них, следует либо менять все значения вместе или не менять их вообще.

Коннасценция идентичности (Connascence of Identity, CoI)

Связанные компоненты должны ссылаться на одну и ту же сущность.

Стандартный пример данного типа коннасценции — два независимых компонента, которые совместно используют и обновляют общую структуру данных, например распределенную очередь.

Динамическую коннасценцию определить сложнее, потому что у нас нет инструментов, чтобы проводить анализ выполняющихся вызовов так же эффективно, как анализ графа вызовов.

Свойства коннасценции

Коннасценция — это инструмент анализа для архитекторов и разработчиков, и некоторые ее свойства чрезвычайно полезны. Эти свойства описаны ниже.

Сила

Сила коннасценции определяется тем, насколько просто проводить ее рефакторинг; очевидно, желательно иметь коннасценции разных типов, как показано на рис. 5.1. Характеристики связанности кодовой базы можно улучшить, проведя рефакторинг с целью определения лучшей коннасценции.

Статическая коннасценция предпочтительнее динамической, поскольку ее можно определить путем простого анализа исходного кода, а благодаря современным инструментам улучшать этот тип коннасценции легко. Рассмотрим пример *коннасценции смысла*, которую можно улучшить с помощью рефакторинга до *коннасценции имени*, введя именованную константу, а не магическое значение.

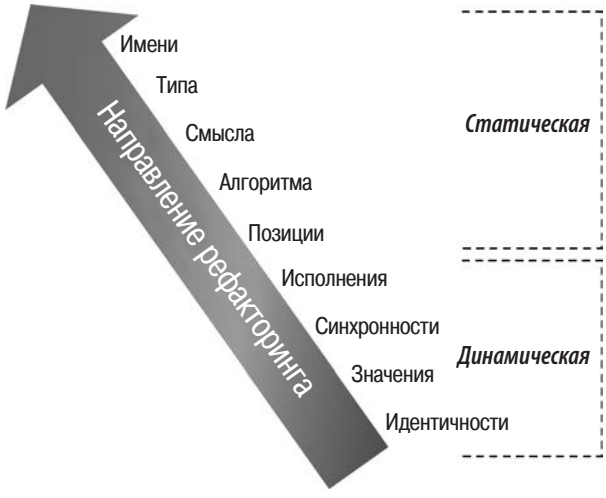


Рис. 5.1. Сила коннасценции обеспечивает качественный рефакторинг

Локальность

Локальность коннасценции измеряется близостью расположения модулей в кодовой базе. В проксимальном коде (расположенном в одном модуле) видов коннасценции обычно больше и они более высокого уровня, чем в разделенном (расположенном в разных модулях или кодовых базах). Другими словами, если коннасценции одного вида находятся на далеком расстоянии друг от друга, то это говорит о плохой связанности, а если на близком — то о нормальной. Например, наличие коннасценции смысла в двух классах одного компонента меньше вредит кодовой базе, чем наличие этой же коннасценции в двух компонентах.

При разработке продуктов необходимо объединять силу и локальность. Качество кода, в котором сильные коннасценции объединены в одном модуле, лучше, чем качество кода, в котором такие же коннасценции разнесены по разным модулям.

Степень

Степень коннасценции относится к мере ее воздействия — влияет ли она только на некоторые классы или на многие? Чем меньше степень коннасценции, тем меньше она вредит кодовой базе. Другими словами, высокая динамическая коннасценция нестрашна, если у вас всего несколько модулей. Но кодовые базы со временем разрастаются, и небольшие проблемы становятся крупными.

Пейдж-Джонс предлагает три рекомендации по использованию коннасценции для повышения модульности систем:

1. Минимизировать общую коннасценцию, разбив систему на инкапсулированные части.
2. Минимизировать все оставшиеся коннасценции, которые пересекают границы инкапсуляции.
3. Максимизировать коннасценцию в границах инкапсуляции.

Легендарный новатор в области программной архитектуры Джим Вейрих (Jim Weirich), который популяризировал концепцию коннасценции, дает два замечательных совета (https://vimeo.com/10837903):

> Правило степени: преобразуйте сильные коннасценции в более слабые.
>
> Правило локальности: по мере увеличения расстояния между программными элементами используйте более слабые коннасценции.

Коннасценции и ограниченный контекст

Книга Эрика Эванса «Domain-Driven Design»[1] оказала глубокое влияние на современное архитектурное мышление. *Предметно-ориентированное проектирование* (domain-driven design, DDD) — это техника моделирования, которая позволяет разделять сложные предметные области. DDD определяет *ограниченный контекст* (bounded context), в котором все, что связано с предметной областью, видно внутри, но непрозрачно для других ограниченных контекстов. Концепция ограниченного контекста признает, что каждая сущность лучше всего работает в локализованном контексте. Таким образом, вместо того чтобы создавать единый класс `Customer` для всей организации, каждая предметная область может создать свой собственный класс и согласовывать различия в точках интеграции. Подобная изоляция распространяется и на другие детали реализации, такие как схемы баз данных, что приводит к изолированию данных в степени, характерной для микросервисов, в основе которых лежит принцип ограниченного контекста.

Одна из целей архитекторов, проектирующих системы по принципам DDD, в том числе модульные монолиты и микросервисы, — предотвратить «утечку» деталей реализации за пределы ограниченного контекста. Это не препятствует взаимодействию ограниченных контекстов, но это взаимодействие ограничивается контрактом (подробнее см. в разделе «Контракты» на с. 134).

Внимательные читатели заметят общее между рекомендацией 1993 года, касающейся *локальности* коннасценций, и рекомендацией 2003 года, касающейся *ограниченного контекста*: расширение области распространения коннасценции приведет к *хрупкости* (brittleness) архитектуры. Хрупкая архитектура — это архитектура, в которой небольшое изменение в одном месте может привести к непредсказуемым и нелокализованным поломкам в других местах.

Например, рассмотрим самую неблагоприятную ситуацию, которая, к сожалению, случается в некоторых архитектурах: раскрытие схемы базы данных приложения в точке интеграции архитектуры. Схема базы данных приложения в рамках DDD является частью ограниченного контекста — деталью реализации. Раскрытие этой детали другим приложениям означает, что изменение в базе данных одного приложения может непредсказуемо нарушить работу других приложений. Таким образом, раскрытие деталей реализации в широком масштабе вредит целостности всей архитектуры.

Общая тенденция в архитектуре, актуальная по крайней мере с 1993 года (а скорее всего, и с более ранних пор), заключается в максимально возможном ограничении связанности реализации — нам просто сложно найти для этого

[1] *Эванс Э.* «Предметно-ориентированное проектирование».

наиболее удачную формулировку. Неважно, как архитекторы называли свои действия — *ограничением контекста* или *соблюдением принципа локальности коннасценций*, — они на протяжении десятилетий старались решить или сгладить проблему связанности.

Хотя ограниченный контекст — это последняя попытка выразить философию эффективной связанности, эта концепция восходит к DDD и связана с ним, а значит, является абстрактной. Нам нужна архитектурная концепция, которая отражает ограниченный контекст, но выражает его в технических терминах и позволяет более тесно согласовать его с проблемами архитектуры (а не абстрактными проблемами проектирования).

Кванты архитектуры и гранулярность

Программные системы связаны между собой разными способами. Как программные архитекторы мы анализируем ПО с разных точек зрения. Но связь между системами обеспечивается не только сцеплением на уровне компонентов. Для реализации многих бизнес-концепций части системы объединяются семантически, формируя *функциональную связность*. Чтобы успешно создавать эволюционные продукты, разработчики должны учитывать *все* точки, связанность в которых может нарушиться.

В физике *квант* — это минимальное количество любой физической сущности, участвующее во взаимодействии. *Квант архитектуры* — это независимо развертываемый компонент с высокой функциональной связностью, который включает все структурные элементы, необходимые для нормального функционирования системы. В монолитной архитектуре квантом является все приложение; все его компоненты сильно сцеплены, поэтому их все необходимо развертывать одновременно.

Термин *«квант»*, конечно, уже широко используется в такой области физики, как *квантовая механика*. Однако авторы этой книги выбрали этот термин по тем же причинам, что и физики. «*Квант*» произошел от латинского слова *quantus*, означающего «сколько». Прежде чем его взяли на вооружение физики, этот термин использовался в юриспруденции для обозначения «обязательной или допустимой суммы» — например, при возмещении ущерба. Этот термин также встречается в математической топологии, описывающей свойства семейств геометрических фигур.

Архитектурный квант служит мерой свойств топологии и поведения в программной архитектуре, связанных с соединением и взаимодействием ее частей.

Статическая связанность

Описывает, как статические зависимости разрешаются в архитектуре с помощью контрактов. Эти зависимости включают операционную систему, фреймворки и/или библиотеки для управления транзитивными зависимостями, а также любые другие эксплуатационные требования для функционирования кванта.

Динамическая связанность

Описывает, как кванты взаимодействуют во время выполнения, синхронно или асинхронно. Это значит, что фитнес-функции для этой характеристики должны быть *непрерывными*, обычно с использованием мониторов.

Определенные здесь *статическая* и *динамическая* связанность аналогичны соответствующим типам коннасценции. Понять разницу меду ними просто: *статическая связанность* описывает, как сервисы *соединены* между собой, а *динамическая* — как сервисы *вызывают* друг друга во время выполнения рабочего процесса. Например, в архитектуре микросервисов сервис должен содержать зависимые компоненты, такие как база данных, и связанность между сервисом и этими компонентами статическая — сервис не может работать без необходимых данных. Этот сервис может *вызывать* другие сервисы в ходе выполнения, и в этом случае между ними возникает *динамическая* связанность. Ни одному из сервисов для выполнения не требуется присутствие другого, только данный рабочий процесс. Таким образом, статическая связанность анализирует эксплуатационные зависимости, а динамическая — зависимости взаимодействия.

Квант архитектуры

Квант архитектуры — это независимо развертываемый артефакт с высокой функциональной связностью, а также высокой статической и синхронной динамической связанностью.

Типичный пример кванта архитектуры — хорошо согласованный микросервис в рамках рабочего процесса.

Определение выше включает важные характеристики; рассмотрим каждую из них подробно, так как они используются для большинства примеров в книге.

Независимое развертывание

Независимое развертывание касается различных свойств архитектурного кванта — каждый квант представляет собой отдельную развертываемую единицу в рамках конкретной архитектуры. Таким образом, монолитная архитектура, которая развертывается как единое целое, по определению представляет собой единый квант архитектуры. В распределенной архитектуре, такой как микро-

сервисы, разработчики стремятся развертывать сервисы независимо, часто с высокой степенью автоматизации. Таким образом, с точки зрения *независимого развертывания*, сервис в архитектуре микросервисов представляет собой квант архитектуры (в зависимости от связанности — см. ниже).

Тот факт, что каждый квант архитектуры представляет собой развертываемый актив, полезен в нескольких целях. Во-первых, границы кванта архитектуры — это то, что имеет значение и для архитекторов, и для разработчиков, и для группы эксплуатации — все они понимают их со своих точек зрения. Архитекторы — с точки зрения характеристик связанности, разработчики — с точки зрения области поведения, а группа эксплуатации — с точки зрения развертываемых компонентов.

Во-вторых, границы квантов отражают одну из сил (статическая связанность), которую архитекторы должны учитывать при обеспечении надлежащей гранулярности сервисов в распределенной архитектуре. Зачастую в микросервисной архитектуре трудно определить, при какой гранулярности сервисов набор компромиссов будет оптимальным. Некоторые компромиссы касаются развертывания: с какой периодичностью необходимо проводить развертывание данного сервиса, на какие еще сервисы оно повлияет, какие приемы разработки использовать и т. д. Архитекторам необходимо четко понимать, где именно пролегают границы развертывания в распределенных архитектурах.

В-третьих, для *независимого развертывания* квант архитектуры должен включать общие точки сцепления (coupling points), такие как базы данных. В большинстве случаев при обсуждении архитектуры не затрагиваются такие вопросы, как базы данных и пользовательские интерфейсы, но в реальных системах они важны. Таким образом, любая система, использующая общую базу данных, не соответствует критерию квантования архитектуры для независимого развертывания, если только база данных не развертывается синхронно с приложением. Многие распределенные системы, в которых можно было бы выделить несколько квантов, не подходят для независимого развертывания, если используют общую базу данных, которая развертывается по отдельному графику. Таким образом, просто определить границы развертывания недостаточно. Чтобы деление на кванты было эффективным, необходимо также учесть второй критерий квантования архитектуры — *сильную функциональную связность*.

Сильная функциональная связность

Сильная функциональная связность (high functional cohesion) в структурном отношении указывает на близость родственных элементов: классов, компонентов, сервисов и т. д. Традиционно в информатике выделяются различные типы

связности на основании общего *модуля*, в качестве которого могут выступать *классы* или *компоненты*, в зависимости от платформы. С точки зрения предметной области техническое определение *сильной функциональной связности* соответствует принципам *ограниченного контекста* в DDD: поведение и данные, реализующие рабочий процесс определенной предметной области.

В терминах *независимого развертывания* гигантская монолитная архитектура, строго говоря, представляет собой один квант архитектуры. Однако она почти наверняка не является функционально целостной, а скорее включает в себя функциональность всей системы. Чем больше монолит, тем меньше вероятность того, что он будет представлять собой единое функциональное целое.

В идеале в архитектуре микросервисов каждый сервис должен моделировать одну предметную область или один рабочий процесс и поэтому должен обладать сильной функциональной связностью. Связность в данном случае характеризует не то, как сервисы взаимодействуют для выполнения работы, а то, насколько независим один сервис от другого и насколько они сцеплены.

Сильная статическая связанность

Сильная статическая связанность (high static coupling) означает, что элементы кванта архитектуры тесно связаны друг с другом, что на самом деле является предметом контрактов. Архитекторы считают такие понятия, как REST и SOAP, разновидностями контрактов, но сигнатуры методов и операционные зависимости (через точки сцепления, такие как IP-адреса и URL) также представляют собой контракты, которые мы рассматриваем в разделе «Контракты» на с. 134.

Статическая связанность отчасти является мерой кванта архитектуры, и эта мера довольно проста в большинстве топологий. Например, диаграммы ниже иллюстрируют стили архитектуры, описанные в книге «Fundamentals of Software Architecture»[1], и их статическую квантовую связанность.

Во всех стилях монолитной архитектуры квантовая мера равна одному.

Как показано на рис. 5.2, любая архитектура, которая развертывается как единое целое и использует одну базу данных, всегда будет иметь один квант. Квант архитектуры, измеряемый статической связанностью, включает базу данных, поэтому система с одной базой данных не может иметь больше одного кванта. Таким образом, измерение кванта архитектуры статической связанностью помогает определить точки сцепления во всей архитектуре, а не только в разраба-

[1] *Ричардс М., Форд Н.* «Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы». СПб., издательство «Питер».

тываемых программных компонентах. В большинстве архитектур предметной области имеется одна точка сцепления — обычно это база данных, которая делает их квантовую меру равной единице.

До сих пор все топологии измерялись *одним* квантом архитектуры. Однако в распределенных архитектурах может быть несколько квантов, хотя и не обязательно. Например, стиль событийно-ориентированной архитектуры, или архитектуры, управляемой событиями (event-driven architecture), в котором используется посредник, всегда будет оцениваться в один квант архитектуры, как показано на рис. 5.3.

На рис. 5.3 представлена модель распределенной архитектуры, но две точки сцепления делают ее одним архитектурным квантом: база данных, как в монолитных архитектурах, описанных выше, а также оркестратор запросов `Request Orchestrator` — любая единая точка сцепления, необходимая для функционирования архитектуры, формирует вокруг себя единый архитектурный квант.

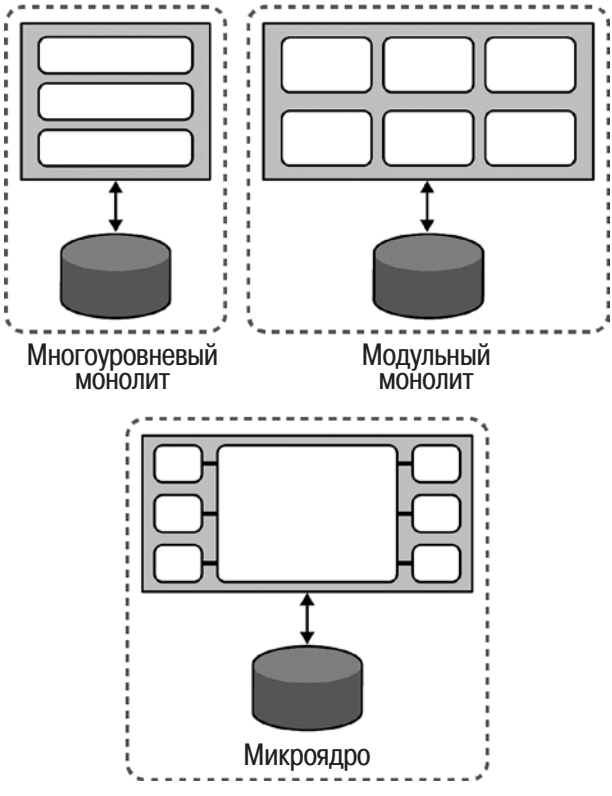


Рис. 5.2. В монолитной архитектуре всегда один квант

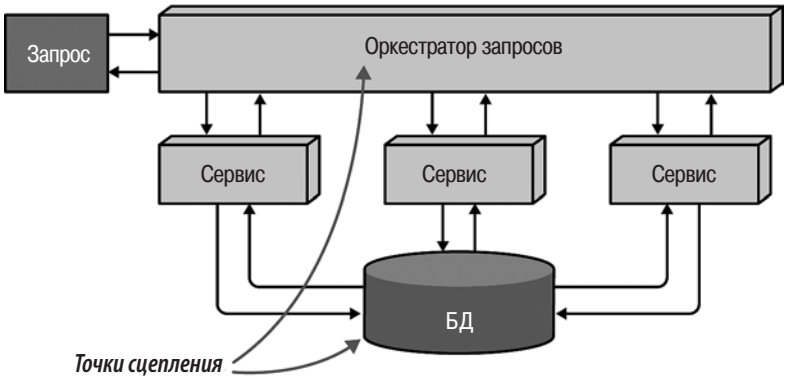


Рис. 5.3. Событийно-ориентированная архитектура с использованием шаблона посредника имеет один квант

Наличие брокеров в событийно-ориентированной архитектуре (обуславливающее отсутствие централизованного посредника) делает ее менее сцепленной, но не гарантирует полного разделения. Рассмотрим событийно-ориентированную архитектуру, показанную на рис. 5.4.

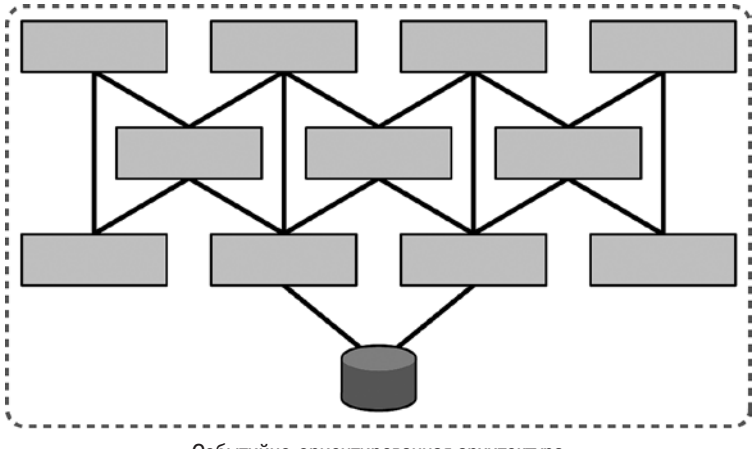


Рис. 5.4. Даже распределенная архитектура, например брокерная событийно-ориентированная архитектура, может представлять собой один квант

На рис. 5.4 представлена модель брокерной событийно-ориентированной архитектуры (без централизованного посредника), которая является единым архитектурным квантом, поскольку все сервисы используют единую

реляционную базу данных, выступающую в качестве общей точки сцепления. Вопрос, на который отвечает *статический анализ* кванта архитектуры, заключается в том, обусловлена ли подобная конфигурация необходимостью самозагрузки сервиса. Даже если в событийно-ориентированной архитектуре некоторые сервисы и не обращаются к базе данных, когда они взаимодействуют с сервисами, которые это делают, они становятся частью статической связанности архитектурного кванта.

Но как быть, когда общих точек сцепления в распределенных архитектурах нет? Рассмотрим модель событийно-ориентированной архитектуры на рис. 5.5. Архитекторы разработали систему с двумя хранилищами данных, в которой нет статических зависимостей между наборами сервисов. Обратите внимание, что любой квант архитектуры может функционировать в рабочей экосистеме. Возможно, он не сможет участвовать во всех ее рабочих процессах, но сможет успешно отправлять запросы и получать их в рамках архитектуры.

Мера статической связанности для кванта архитектуры оценивает зависимости связанности архитектурных и операционных компонентов. Таким образом, операционная система, хранилище данных, брокер сообщений, оркестрация контейнеров и все остальные операционные зависимости образуют точки статического сцепления архитектурного кванта с использованием максимально строгих контрактов (подробнее о роли контрактов в архитектурных квантах см. в разделе «Контракты» на с. 134).

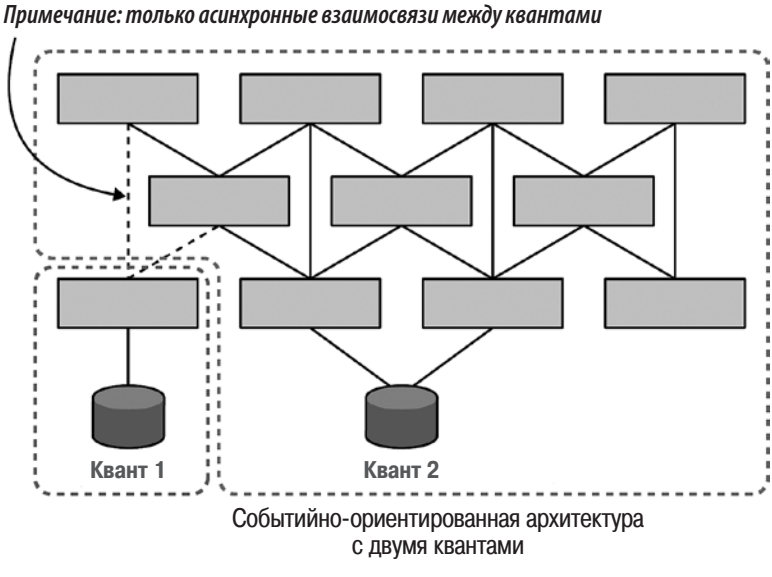


Рис. 5.5. Событийно-ориентированная архитектура с двумя квантами

Для архитектуры микросервисов характерна высокая степень обособленности сервисов, включая зависимости в данных. В таких моделях архитекторы предпочитают обеспечивать высокую степень разделения и стараются не создавать точек сцепления сервисов, чтобы каждый сервис представлял собой отдельный квант, как показано на рис. 5.6.

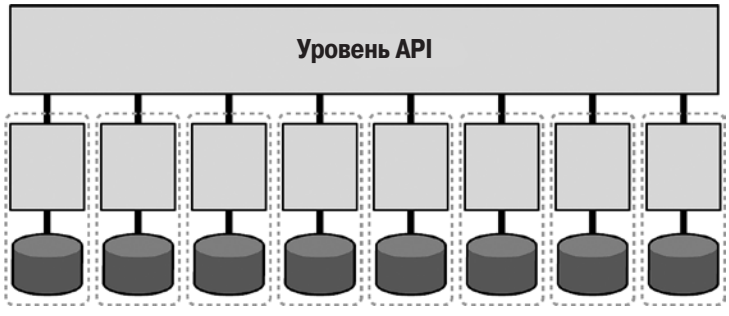


Рис. 5.6. Микросервисы могут образовывать собственные кванты

На рис. 5.6 каждый сервис (действующий как ограниченный контекст) обладает собственным набором характеристик архитектуры — один сервис может иметь более высокую масштабируемость или безопасность, чем другой. Такое гранулярное распределение характеристик архитектуры является одним из преимуществ архитектуры микросервисов. Благодаря высокой степени разделения сервисов команды разработчиков могут работать максимально быстро и не беспокоиться о нарушении других зависимостей.

Однако если система сильно сцеплена с пользовательским интерфейсом, то архитектура образует единый квант, как показано на рис. 5.7.

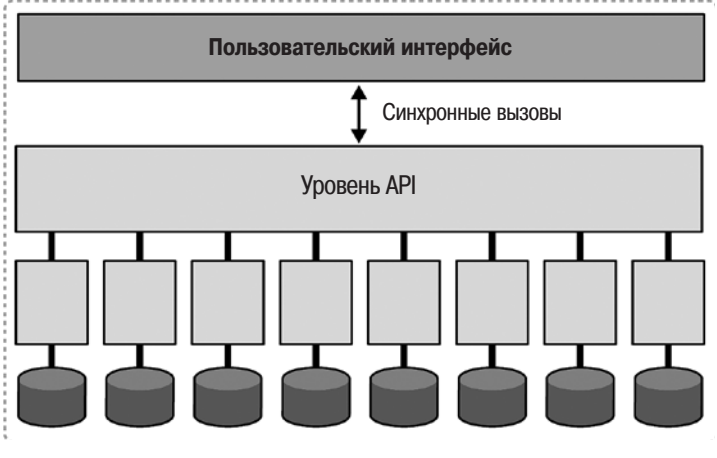


Рис. 5.7. Сильная связанность с пользовательским интерфейсом приводит к тому, что в архитектуре микросервисов имеется всего один квант

Традиционно пользовательские интерфейсы выступают точками сцепления между фронтендом и бэкендом, и большинство пользовательских интерфейсов не будет работать, если бэкенд недоступен.

Кроме того, сложно добиться различия характеристик операционной архитектуры (производительности, масштаба, эластичности, надежности и т. д.) в каждом сервисе, если все они должны взаимодействовать в одном пользовательском интерфейсе (особенно в случае синхронных вызовов, рассматриваемых в разделе «Динамическая квантовая связанность» на с. 130).

Архитекторы разрабатывают асинхронные пользовательские интерфейсы, в которых отсутствует связанность между фронтендом и бэкендом. Во многих современных проектах микросервисов используется фреймворк *микрофронтенда* для элементов пользовательского интерфейса. В такой архитектуре элементы интерфейса, которые взаимодействуют от имени сервисов, генерируются самими сервисами. Поверхность пользовательского интерфейса выступает в качестве холста, на котором появляются элементы интерфейса, а также облегчает взаимодействие слабо сцепленных компонентов, обычно с использованием событий. Подобная архитектура показана на рис. 5.8.

На рис. 5.8 четыре заштрихованных сервиса и соответствующие им микрофронтенд-элементы образуют архитектурные кванты: характеристики архитектуры каждого из этих сервисов могут быть различными.

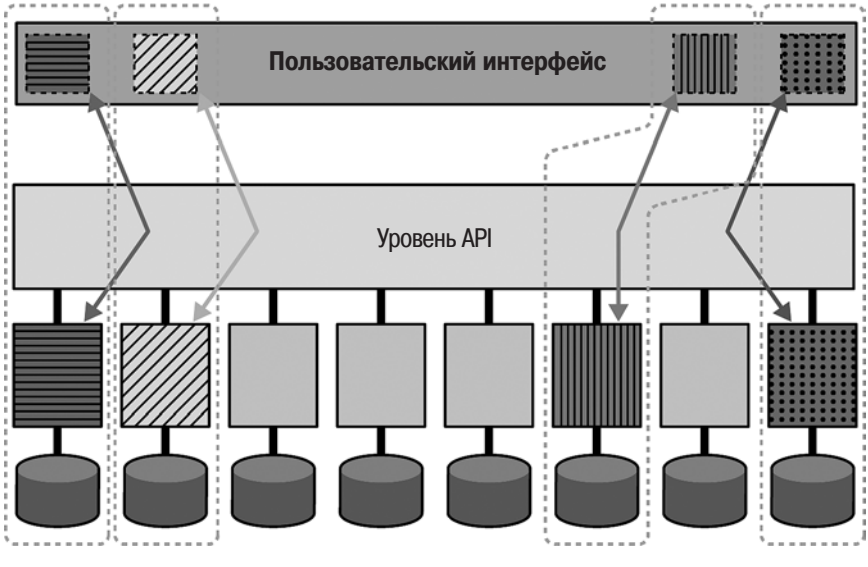


Рис. 5.8. В архитектуре микрофронтенда каждый сервис и компонент пользовательского интерфейса образуют архитектурный квант

Любая точка сцепления в архитектуре может являться статической с точки зрения квантования. Например — база данных, совместно используемая двумя системами, как показано на рис. 5.9.

Статическая связанность системы представляет ценность для аналитики даже в сложных системах с интеграционной архитектурой. Все чаще для понимания унаследованной архитектуры используются статические квантовые диаграммы, изображающие, как части системы соединены вместе, что помогает определить, какие из них затронут изменения и как можно при необходимости разделить архитектуру.

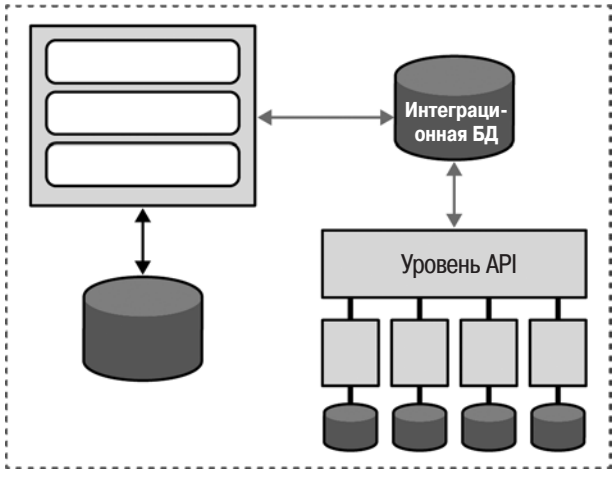


Рис. 5.9. Общая база данных образует точку сцепления между двумя системами, формируя единый квант

Статическая связанность — только одна из сил, действующих в распределенных архитектурах; вторая — это *динамическая* связанность.

Динамическая квантовая связанность

Последняя часть определения архитектурного кванта касается синхронной связанности во время выполнения — другими словами, поведения архитектурных квантов при их взаимодействии друг с другом для формирования рабочих процессов в распределенной архитектуре.

Природа того, *как* сервисы вызывают друг друга, обуславливает трудность выбора компромиссов, поскольку возможные решения лежат в нескольких плоскостях и на них влияют три взаимосвязанные силы.

Взаимодействие

Определяет тип синхронизации вызова: *синхронный* или *асинхронный*.

Согласованность

Определяет, должен ли быть рабочий процесс взаимодействия атомарным или можно применять согласованность в конечном счете, или конечную согласованность (eventual consistency).

Координация

Описывает, использует ли рабочий процесс *оркестратор* или службы взаимодействуют по паттерну *хореографии*.

Взаимодействие

Один из ключевых вопросов для архитектора, выстраивающего взаимодействие между двумя сервисами, заключается в том, каким должно быть это взаимодействие — *синхронным* или *асинхронным*.

Синхронное взаимодействие предполагает, что отправитель запроса будет ждать ответа от получателя, как показано на рис. 5.10.

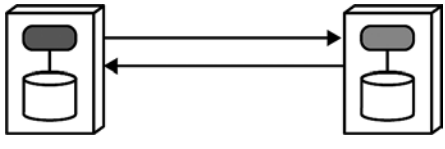


Рис. 5.10. Синхронный вызов ожидает ответа от получателя

На рис. 5.10 вызывающая служба совершает вызов (используя один из протоколов, поддерживающих синхронные вызовы, например gRPC) и *блокирует* его (не выполняет дальнейшую обработку), пока получатель не вернет некоторое значение (или статус, указывающий на изменение состояния или ошибку).

Асинхронное взаимодействие между двумя службами заключается в том, что вызывающая сторона передает сообщение получателю (обычно через определенный механизм, например очередь сообщений) и после получения подтверждения, что сообщение будет обработано, продолжает свою работу. Если запрос требует возврата значения, получатель может использовать очередь ответов для (асинхронного) уведомления вызывающей стороны о результате, как показано на рис. 5.11.

На рис. 5.11 вызывающая сторона помещает сообщение в очередь и продолжает обработку, пока не получит уведомление от получателя, что запрошенная информация доступна через обратный вызов. Обычно для реализации асинхронного

взаимодействия архитекторы используют очереди сообщений (изображаемые на схемах в виде труб, наложенных на стрелки, обозначающие взаимодействие), и поскольку они используются часто и перегружают картинку, на диаграммах их просто опускают. И конечно, асинхронное взаимодействие можно реализовать и без очередей сообщений, используя различные библиотеки или фреймворки. На обеих диаграммах на рис. 5.11 изображена асинхронная передача сообщений, но нижняя диаграмма визуально менее подробная и показывает меньше деталей реализации.

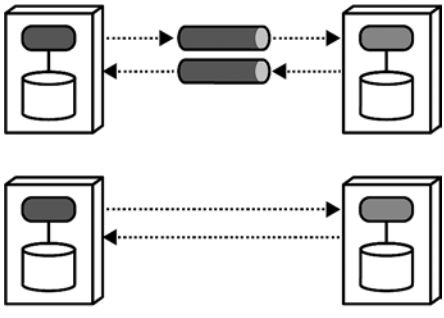


Рис. 5.11. Параллельная обработка при асинхронном взаимодействии

При выборе способа взаимодействия сервисов архитекторы должны учитывать ряд существенных компромиссов. Принятое решение будет влиять на синхронизацию, обработку ошибок, транзакционность, масштабируемость и производительность. В оставшейся части этой книги обсуждаются многие из этих вопросов.

Согласованность

Согласованность (consistency) означает строгую целостность транзакций, которая должна соблюдаться в вызовах. Атомарные транзакции (транзакции «все или ничего», требующие согласованности *во время* обработки запроса) находятся на одной стороне спектра, а различные степени согласованности в конечном счете — на другой.

Транзакционность — участие нескольких сервисов в транзакции «все или ничего» — одна из самых сложных проблем для моделирования в распределенных архитектурах, поэтому в общем случае рекомендуется избегать кросс-сервисных транзакций. Эта сложная тема рассматривается в книге «Software Architecture: The Hard Parts»[1] (O'Reilly) и выходит за рамки данного издания.

[1] *Форд Н.* и др. «Современный подход к программной архитектуре: сложные компромиссы». СПб., издательство «Питер».

Координация

Иногда важно также определить степень *координации* для рабочего процесса, моделируемого взаимодействием. Существует два общеприятых типовых паттерна для микросервисов — *оркестрация* и *хореография*. Если рабочий процесс несложен — один сервис, отвечающий на запрос, — вопрос координации не особенно актуален. Однако по мере роста сложности рабочих процессов возрастает и потребность в координации.

Все три фактора — взаимодействие, согласованность и координация — определяют выбор архитектурного решения. Однако, что очень важно, этот выбор нельзя сделать, приняв во внимание лишь один из них: каждый фактор оказывает гравитационное воздействие на другие. Например, транзакционность выше в синхронных архитектурах с оркестрацией, а масштабируемость — в асинхронных хореографичных системах с согласованностью в конечном счете.

Если рассматривать эти силы как связанные друг с другом, то они образуют трехмерную структуру, показанную на рис. 5.12.

На рис. 5.12 каждый фактор, определяющий взаимодействие сервисов, представлен в виде измерения. Для конкретного решения архитектор может построить пространственный график, отражающий силу этих факторов. На практике полезно создавать матрицы для представления о том, какое влияние может оказать изменение любого из этих взаимосвязанных факторов.

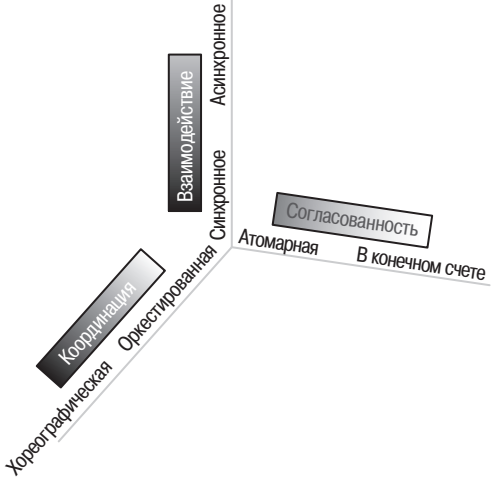


Рис. 5.12. Измерения динамической квантовой связанности

Контракты

Одним из факторов, затрагивающих практически все области программной архитектуры и влияющих на все аспекты принятия архитектурных решений, являются *контракты,* в широком смысле определяемые как то, что связывает разрозненные части архитектуры. Словарное определение контракта звучит так:

Контракт

> Письменное или устное соглашение, чаще всего касающееся сфер трудоустройства, продажи или аренды, условия которого определяются законом.

В разработке ПО контракты широко используются для таких понятий, как точки интеграции в архитектуре, и существует множество разновидностей контрактов: SOAP, REST, gRPC, XML-RPC и прочие аббревиатуры. Однако мы расширяем это определение и делаем его более последовательным:

Контракт

> Формат, используемый частями архитектуры для передачи информации или зависимостей.

Это определение *контракта* охватывает все методы, используемые для соединения частей системы, включая транзитивные зависимости для фреймворков и библиотек, внутренние и внешние точки интеграции, кэши и любые другие схемы взаимодействия.

Архитектурные контракты варьируются от строгих до свободных, как показано на рис. 5.13.

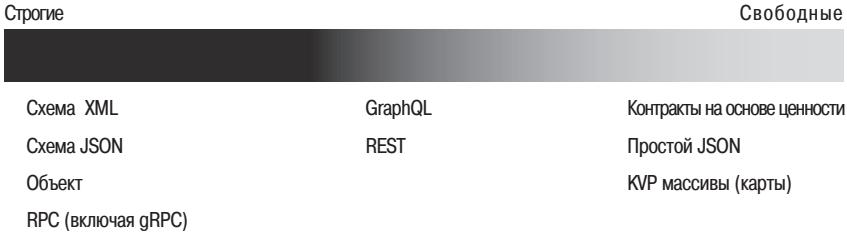


Рис. 5.13. Спектр типов контрактов от строгих к свободным

На рис. 5.13 для наглядности показаны несколько типов контрактов. Строгий контракт требует однозначного соблюдения имен, типов, порядка и остальных деталей. Примером самого строгого из возможных контрактов в разработке является удаленный вызов метода с использованием механизма RMI в Java. В этом случае удаленный вызов имитирует вызов внутреннего метода с одинаковым именем, параметрами, типами и остальными деталями.

Многие строгие форматы контрактов имитируют семантику вызовов методов. Например, существует множество разнообразных протоколов RPC, аббревиатуры для *Remote Procedure Call (удаленный вызов процедур)*. gRPC (https://grpc.io/) — пример популярного фреймворка удаленного вызова, в котором по умолчанию используются строгие контракты.

Многие архитекторы предпочитают строгие контракты, поскольку они моделируют семантическое поведение вызовов внутренних методов. Однако строгие контракты делают архитектуру интеграции хрупкой, чего следует избегать. Как обсуждается в разделе «Паттерны повторного использования» на с. 143, элементы, которые одновременно и часто изменяются, и используются несколькими частями архитектуры, являются источником проблем; контракты подходят под это описание, поскольку они образуют клей в распределенной архитектуре: чем чаще их необходимо изменять, тем больше проблем для других сервисов это создает. Однако строгие контракты использовать необязательно: это необходимо делать только тогда, когда это приносит пользу.

Даже для такого условно свободного формата, как JSON (https://www.json.org/json-en.html), существуют способы выборочного добавления информации о схеме к простым парам имя/значение. В примере 5.1 показан строгий контракт JSON с добавленной информацией о схеме.

Пример 5.1. Строгий контракт JSON

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
      "acct": {"type": "number"},
      "cusip": {"type": "string"},
      "shares": {"type": "number", "minimum": 100}
   },
    "required": ["acct", "cusip", "shares"]
}
```

В примере 5.1 первая строка ссылается на определение схемы, которую мы используем и которой необходимо соответствовать. Мы определяем три свойства: `acct`, `cusip` и `shares`, а также их типы и, в последней строке, какие из них являются обязательными. Таким образом создается строгий контракт с указанием обязательных полей и типов.

Примерами более свободных контрактов являются такие форматы, как REST и GraphQL (https://graphql.org/), совершенно разные по своей сути, но имеющие более слабую связанность, чем форматы на основе RPC. В *REST* архитектор моделирует ресурсы, а не конечные точки методов или процедур, поэтому такой контракт менее хрупкий. Например, если REST-ресурс описывает детали конструкции самолета и поддерживает запросы о сиденьях, то этот запрос не

сломается при добавлении в ресурс информации о двигателях — добавление дополнительной информации не нарушит имеющуюся конструкцию.

Аналогичным образом *GraphQL* в распределенных архитектурах используется для предоставления агрегированных данных только для чтения, а не для выполнения дорогостоящих оркестрированных вызовов различных сервисов. В примерах 5.2 и 5.3 рассмотрены два представления GraphQL, содержащих разные, но рабочие контракты `Profile`.

Пример 5.2. Представление Wishlist Profile клиента

```
type Profile {
    name: String
}
```

Пример 5.3. Представление Profile клиента

```
type Profile {
    name: String
    addr1: String
    addr2: String
    country: String
    . . .
}
```

Концепция *профиля* (profile) появляется в обоих примерах, 5.2 и 5.3, но с разными значениями. В этом сценарии `Wishlist` клиента не имеет внутреннего доступа к имени клиента, только к уникальному идентификатору. Таким образом, ему необходим доступ к `Profile` клиента, который сопоставляет идентификатор с именем клиента. `Profile` клиента, помимо имени, включает в себя большое количество информации о клиенте. Единственное, что необходимо `Wishlist` в `Profile`, — это имя.

Распространенный антипаттерн, на чью удочку попадаются некоторые архитекторы, заключается в предположении, что `Wishlist` может понадобиться всем остальным частям, поэтому его включают в контракт с самого начала. Это пример Stamp Coupling (связанности по образцу) (https://wiki.c2.com/?StampCoupling), и в большинстве случаев она является антипаттерном, поскольку разрушает то, что разрушать не нужно, делая архитектуру хрупкой и не принося при этом никакой пользы. Например, если `Wishlist` требует только имя клиента из `Profile`, но в контракте указано каждое поле `Profile` (на всякий случай), то изменение `Profile`, не важное для `Wishlist`, приводит к нарушению контракта и координации, которое придется исправлять.

Чтобы сохранить баланс между семантической связанностью и требуемой информацией, но при этом не делать интеграционную архитектуру излишне хрупкой, целесообразно поддерживать контракты на уровне «необходимо знать».

На другом конце спектра связанности контрактов находятся максимально свободные контракты, часто представляющие собой пары имя/значение в таких форматах, как YAML (https://yaml.org/) и JSON, как показано в примере 5.4.

Пример 5.4. Пары имя/значение в JSON

```
{
  "name": "Mark",
  "status": "active",
  "joined": "2003"
}
```

В этом примере мы видим только непосредственные факты. Никаких дополнительных метаданных, информации о типе и прочего, только пары имя/значение.

Использование свободных контрактов позволяет создавать максимально разделенные системы, что часто является целью таких архитектур, как микросервисы. Однако свобода подразумевает компромиссы, в том числе недостаточную определенность, верификацию и усложнение логики приложения. Во многих случаях контракты теперь заменяются фитнес-функциями.

Практический пример: микросервисы как эволюционная архитектура

Архитектура микросервисов определяет физически ограниченные контексты между архитектурными элементами, инкапсулирующие все части, которые могут меняться. Этот тип архитектуры обеспечивает возможность внедрения инкрементных изменений. В архитектуре микросервисов ограниченный контекст служит границей квантов и включает зависимые компоненты, такие как серверы баз данных. Он также может включать, к примеру, средства поиска и инструменты отчетности — все компоненты, обеспечивающие функциональность сервиса, как показано на рис. 5.14.

На рис. 5.14 сервис включает компоненты кода, сервера базы данных и поиска. Принцип ограничения контекста в микросервисах предусматривает совместную эксплуатацию всех частей сервиса с опорой на современные практики DevOps. В следующем разделе мы разберем некоторые распространенные архитектурные паттерны и стандартные границы квантов в них.

Традиционно несвязанные роли, такие как архитектор и инженер группы эксплуатации, в эволюционной архитектуре должны работать совместно. Архитектура является абстрактной, пока она не эксплуатируется; разработчики должны учитывать, как ее компоненты сочетаются в реальных условиях. Независимо от выбранного паттерна архитектуры необходимо также явно задать размер кванта.

Малые кванты подразумевают более быстрые изменения. Как правило, с ними легче работать, чем с большими. Размер кванта определяет нижнюю границу возможных инкрементных изменений в архитектуре.

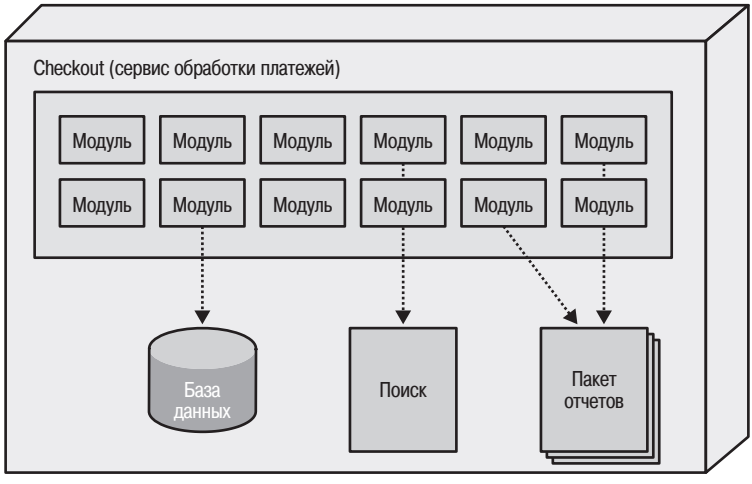


Рис. 5.14. Квант архитектуры микросервиса включает сервис и все его зависимые части

В основе философии микросервисной архитектуры лежит объединение практики непрерывной доставки с физическим разделением ограниченных контекстов, а также концепция архитектурных квантов.

В многоуровневой архитектуре основное внимание уделяется *технической* стороне, или механике приложения: персистентности, пользовательскому интерфейсу, бизнес-правилам и так далее. Такой акцент справедлив для большинства архитектур. Однако существует и другая перспектива. Предположим, что одним из ключевых ограниченных контекстов в приложении является *Checkout* (обработка платежей). Где он находится в многоуровневой архитектуре? Концепции предметной области, такие как *Checkout,* распределены по всем слоям архитектуры. Поскольку слои в архитектуре выделены по техническому принципу, в ней нет четкого измерения *предметной области,* как показано на рис. 5.15.

На рис. 5.15 часть контекста *Checkout* существует в пользовательском интерфейсе, другая часть — на уровне бизнес-правил, а его персистентность обрабатывается нижними слоями. Поскольку многоуровневая архитектура не учитывает концепции предметной области, то, чтобы внести изменения в предметную область, разработчикам приходится менять каждый слой. С точки зрения предметной области эволюционность многоуровневой архитектуры равна нулю. В архитектурах с сильной связанностью вносить изменения трудно, поскольку связь между частями, которые необходимо изменить, очень тесная. Тем не менее

в большинстве проектов чаще всего изменяются именно концепции предметной области. Если команды разработчиков организованы по принципу того, за какой уровень архитектуры они отвечают, то, чтобы внести изменения в *Checkout,* потребуется скоординировать много команд.

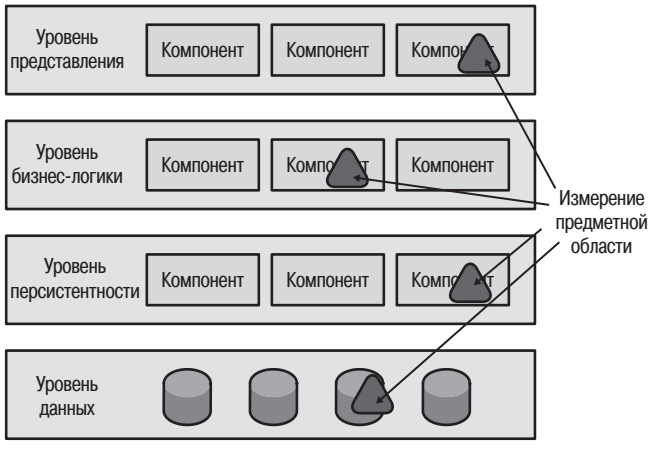


Рис. 5.15. Измерение предметной области встроено в техническую архитектуру

Для сравнения рассмотрим архитектуру, разделенную измерениями *предметной области*, как показано на рис. 5.16.

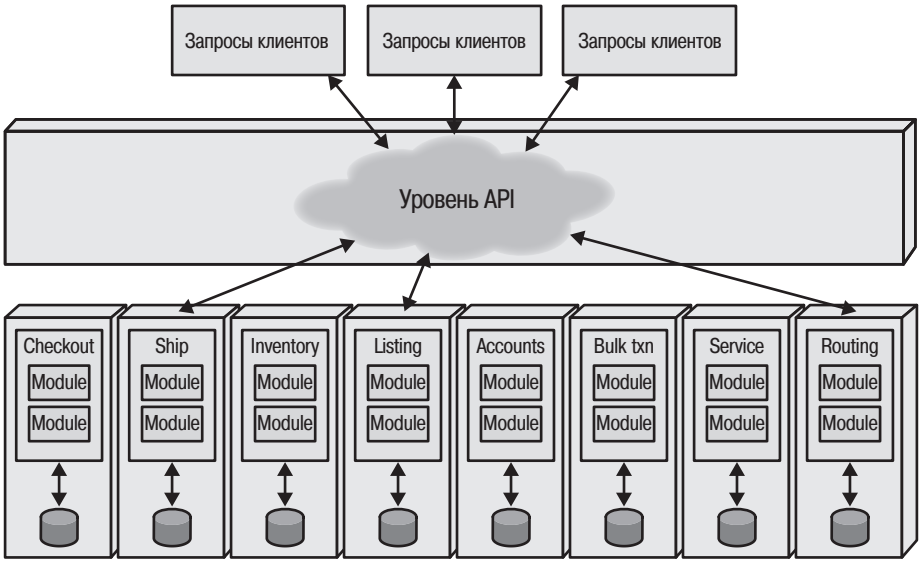


Рис. 5.16. Границами микросервисов служат границы предметных областей со встроенной технической архитектурой

Как показано на рис. 5.16, каждый сервис определяется по принципам DDD путем инкапсулирования технической архитектуры и остальных зависимых компонентов (например, баз данных) в ограниченном контексте, и таким образом создается высокоразделенная архитектура. Каждый сервис «владеет» всеми частями своего ограниченного контекста и взаимодействует с другими ограниченными контекстами посредством обмена сообщениями (например, REST или очереди сообщений). Таким образом, ни один сервис не знает детали реализации другого сервиса (например, схемы баз данных), что позволяет избежать излишней связанности. Эксплуатационная цель подобной архитектуры — заменять один сервис другим, не нарушая работу остальных сервисов.

В основе архитектур микросервисов обычно лежат 7 принципов, описанных в книге «Building Microservices Architectures»[1]:

Моделирование вокруг предметной области бизнес-логики

При разработке микросервисов основное внимание уделяется потребностям бизнеса, а не технической стороне. Таким образом, квант отражает ограниченный контекст. Некоторые разработчики ошибочно полагают, что ограниченный контекст представляет собой отдельную сущность, например `Customer`; однако он представляет собой бизнес-контекст и/или рабочий процесс, например `CatalogCheckout`. Цель микросервисов не в том, чтобы максимально уменьшить размеры сервисов, а в том, чтобы создать полезный ограниченный контекст.

Сокрытие деталей реализации

Техническая архитектура микросервисов инкапсулирована в границах сервиса, определенных бизнес-логикой предметной области. Каждая предметная область образует физически ограниченный контекст. Сервисы интегрируются друг с другом путем передачи сообщений или ресурсов, а не путем раскрытия деталей, таких как схемы баз данных.

Культура автоматизации

Архитектуры микросервисов поддерживают непрерывную доставку, используя пайплайны развертывания для тщательного тестирования кода и автоматизации таких задач, как выделение и развертывание машин. Автоматизированное тестирование особенно актуально для быстро меняющихся сред.

Высокая степень децентрализации

Микросервисы образуют *неразделяемую* архитектуру (shared nothing) для максимального уменьшения связанности. Как правило, дублирование предпочтительнее связанности. Например, в обоих сервисах, `CatalogCheckout`

[1] *Ньюмен С.* «Создание микросервисов». СПб., издательство «Питер».

и `ShipToCustomer`, есть компонент под названием `Item`. Поскольку оба они имеют одинаковое имя и схожие свойства, разработчики стремятся повторно использовать этот компонент в обоих сервисах, думая, что это сэкономит время и усилия. Однако все обстоит с точностью до наоборот, поскольку изменения теперь необходмо распространить на все сервисы, которые совместно используют этот компонент. И всякий раз, изменяя сервис, разработчики должны вносить изменения в совместно используемый компонент. С другой стороны, если каждый сервис имеет свой собственный `Item` и передает необходимую информацию от `CatalogCheckout` к `ShipToCustomer` без связанности с этим компонентом, он может изменяться независимо.

Независимое развертывание

Разработчики и группа эксплуатации ожидают, что каждый компонент сервиса будет развернут независимо от других сервисов (и другой инфраструктуры), что отражает физическую сторону ограниченного контекста. Возможность развернуть один сервис, не затрагивая другие, является одним из ключевых преимуществ этого архитектурного стиля. Более того, разработчики обычно автоматизируют все задачи развертывания и эксплуатации, включая параллельное тестирование и непрерывную доставку.

Изолирование сбоев

Разработчики изолируют сбои как в контексте микросервиса, так и в координации сервисов. Ожидается, что каждый сервис будет обрабатывать разумные сценарии ошибок и по возможности проводить восстановление. В микросервисах часто используются лучшие практики DevOps (например, паттерн Circuit Breaker (https://martinfowler.com/bliki/CircuitBreaker.html), переборки и т. д.). Многие архитектуры микросервисов разрабатываются с соблюдением Реактивного манифеста (Reactive Manifesto) (http://www.reactivemanifesto.org/) — принципов эксплуатации и координации для создания более надежных систем.

Высокая просматриваемость

Проводить ручной мониторинг сотен или тысяч сервисов невозможно (сколько многоадресных SSH-сессий в одном терминале может отследить один разработчик?). Таким образом, задачи мониторинга и журналирования выходят на первый план в этой архитектуре. Если группа эксплуатации не может отследить какой-то сервис, возможно, его не существует.

Основное предназначение микросервисов — изолирование предметной области в физически ограниченном контексте с фокусом на лучшее понимание ее задач. Таким образом, сервис представляет собой квант архитектуры, что делает эту модель отличным примером эволюционной архитектуры. Если один сервис

эволюционирует, чтобы изменить свою базу данных, этот процесс не затрагивает другие сервисы, поскольку ни один из них не знает детали реализации, такие как схемы. Конечно, разработчики изменяющегося сервиса должны будут предоставить эту информацию через точку интеграции между сервисами (надеемся, защищенную фитнес-функцией, такой как *контракты, ориентированные на потребителя* (consumer-driven contracts)), и разработчики вызывающего сервиса вообще не будут знать о произошедших изменениях.

Учитывая, что микросервисы — это образцовый пример эволюционной архитектуры, неудивительно, что они достаточно перспективны с эволюционной точки зрения.

Инкрементное изменение

В архитектурах микросервисов принципы инкрементных изменений просты. Каждый сервис формирует ограниченный контекст вокруг концепции предметной области, что позволяет легко осуществлять изменения, затрагивающие только этот контекст. Архитектуры микросервисов в значительной степени основаны на принципах автоматизации *непрерывной доставки* с использованием пайплайнов развертывания и современных методов DevOps.

Управляемое изменение с помощью фитнес-функций

Для архитектур микросервисов легко создавать как атомарные, так и комплексные фитнес-функции. Каждый сервис имеет четко определенные границы, что позволяет проводить тестирование разного уровня внутри компонентов сервиса. Сервисы должны координироваться путем интеграции, что также требует тестирования. К счастью, сложные методы тестирования совершенствовались параллельно с развитием микросервисов.

Если их преимущества настолько очевидны, то почему микросервисы не стали использоваться гораздо раньше? Дело в том, что автоматическое выделение машин прежде было невозможно. Хотя существовали виртуальные машины (VM), они часто создавались вручную, и процесс этот был довольно длительным. Операционные системы были коммерческими и лицензируемыми и ограниченно поддерживали возможности автоматизации. На архитектуру влияют реальные условия, в частности ограничения бюджета, и это одна из причин, по которой разработчики создают все более сложные архитектуры, совместно использующие ресурсы и разделенные на технических уровнях. Если эксплуатация неудобная и обходится дорого, архитектура создается вокруг нее, как сервис-ориентированные архитектуры, управляемые служебной шиной предприятия.

Continuous Delivery и DevOps стали новым словом в обеспечении динамического равновесия. Теперь описание инфраструктуры «живет» в системе контроля

версий и поддерживает экстремальную автоматизацию [1]. В пайплайнах развертывания можно параллельно запускать несколько тестовых сред для поддержки безопасного непрерывного развертывания. Поскольку большая часть программ имеет открытый исходный код, все меньшее влияние на архитектуру оказывают лицензирование и другие связанные с этим вопросы. Сообщество отреагировало на новые возможности разработки и с их помощью может создавать архитектуру, более ориентированную на предметные области.

В архитектуре микросервисов предметная область инкапсулирует техническую и другие архитектуры, что облегчает эволюцию измерений предметной области. Ни одна точка зрения на архитектуру не является единственно верной, а скорее отражает цели, которые разработчики ставят перед своими проектами. Если создавать измерения с упором исключительно на техническую архитектуру, то изменять их будет проще. Однако если игнорировать сферу предметной области, то созданное измерение будет способно эволюционировать не лучше, чем большой комок грязи.

Паттерны повторного использования

Многократно используемые фреймворки и библиотеки, созданные членами сообщества, обычно с открытым исходным кодом и находящиеся в свободном доступе, представляют большую ценность для нашей отрасли. Очевидно, что возможность повторного использования кода — это хорошо. Однако, как это обычно бывает с хорошими идеями, в какой-то момент ею начинают злоупотреблять, и это создает проблемы. Каждая корпорация стремится повторно использовать код, поскольку ПО кажется таким же модульным, как компоненты электроники. Однако по-настоящему модульный софт до сих пор остается постоянно ускользающей призрачной мечтой.

> Повторное использование кода больше похоже на пересадку органов, чем на соединение блоков Lego.
>
> *— Джон Д. Кук (John D. Cook)*

Хотя разработчики языков уже давно обещают нам блоки Lego, мы все еще пересаживаем органы. Повторное использование — сложная задача, и она не должна выполняться автоматически. Многие менеджеры оптимистично считают, что любой код, написанный разработчиками, можно использовать повторно, но это не всегда так. Ряд компаний старались создать код, который действительно подходит для повторного использования, и им это удалось, но это требует тща-

[1] Подразумевается концепция «инфраструктура как код», то есть набор команд и конфигураций для развертывания виртуальной машины/контейнера. — *Примеч. науч. ред.*

тельного планирования и усилий. Часто разработчики тратят много времени, пытаясь построить многократно используемые модули, которые оказываются практически не пригодными для этого.

В сервис-ориентированных архитектурах (COA) существовала практика искать похожий код и повторно его использовать везде, где только возможно. Например, представим, что в компании есть два контекста: `Checkout` (обработка платежей) и `Shipping` (отгрузка товара). Архитекторы COA замечают, что оба контекста включают понятие `Customer` (клиент). Это побуждает их объединить обоих клиентов в один общий сервис `Customer`, связав с ним `Checkout` и `Shipping`. Целью архитекторов COA было достижение максимальной *каноничности* (canonicality) — когда каждая концепция имеет свое (совместно используемое) расположение.

Как ни странно, чем усерднее разработчики стараются сделать код переиспользуемым, тем сложнее этим кодом пользоваться. Переиспользуемость подразумевает наличие опций и точек принятия решений для различных вариантов применения. Чем больше обработчиков добавляется в код для обеспечения возможности его повторного использования, тем больше это вредит базовому *удобству использования* (usability) кода.

Чем более код переиспользуемый, тем менее он удобен.

Другими словами, простота использования кода часто обратно пропорциональна тому, насколько этот код многоразовый. В такой код нужно добавить функции, предусматривающие множество способов, которыми в конечном итоге он будет использоваться. Это усложняет использование кода для единственной цели.

Микросервисы отказываются от повторного использования кода, придерживаясь принципа *лучше дублирование, чем связанность*: повторное использование подразумевает связанность, а архитектуры микросервисов максимально разделены. Однако цель микросервисов не в дублировании, а в изолировании сущностей внутри предметной области. Сервисы, имеющие общий класс, уже не являются независимыми. В архитектуре микросервисов службы `Checkout` и `Shipping` будут иметь собственное внутреннее представление `Customer`. Если им нужно совместно обрабатывать информацию, связанную с клиентом, они отправляют ее друг другу. Архитекторы не пытаются согласовать и объединить разделенные версии `Customer`. Преимущества повторного использования неочевидны, а связанность, которая при этом возникает, имеет свои недостатки. Таким образом, хотя архитекторы знают об отрицательных сторонах дублирования, они компенсируют их отсутствием ущерба, который мог быть нанесен архитектуре сильной связанностью.

Повторное использование кода не только создает преимущества, но и накладывает дополнительную ответственность. Убедитесь, что точки сцепления, добавленные в код, не нарушают остальную архитектуру. Например, в архитектурах микросервисов для связывания частей сервисов обычно используется сервисная сетка, или сеть микросервисов (service mesh), которая помогает объединить конкретные архитектурные задачи, такие как мониторинг или журналирование.

Эффективное повторное использование = = абстракция + низкая волатильность

Задача, актуальная для многих архитекторов, заключается в том, как согласовать требования двух разных целей организации: комплексного повторного использования и изоляции с помощью ограниченных контекстов на основе DDD. Крупные организации по понятным причинам стремятся по максимуму внедрить в своей экосистеме повторное использование — чем больше они могут повторно использовать, тем меньше им приходится писать с нуля. Однако повторное использование создает связанность, которую многие архитекторы стараются избежать, особенно связанности значительно удаленных модулей.

Sidecar и Service Mesh: ортогональная операционная связанность

Одна из целей проектирования микросервисов — обеспечение высокой степени разделения, что часто выражается фразой «Лучше дублирование, чем связанность». Допустим, два сервиса PenultimateWidgets должны передавать клиентскую информацию, но ограниченный контекст предметно-ориентированного проектирования требует от сервиса оставлять детали реализации приватными. Решение — позволить каждому сервису иметь собственное внутреннее представление сущностей, таких как `Customer`, и передавать эту информацию способами со слабой связанностью, например в виде пар имя/значение в JSON. Обратите внимание, что в этом случае каждый сервис может изменять свое внутреннее представление по собственному усмотрению, включая технологический стек, не нарушая при этом интеграции. Архитекторы обычно скептически относятся к дублированию кода, поскольку оно приводит к проблемам синхронизации, семантическому дрейфу и другим, но иногда существуют бóльшие риски, чем проблемы дублирования, и связанность в микросервисах — как раз один из таких рисков. Таким образом, в архитектуре микросервисов ответ на вопрос «дублировать или связать с уже имеющейся функцией», скорее всего, будет *дублировать, в* то время как в другой архитектуре, например в архитектуре на основе сервисов, — *связать*. Все зависит от ситуации!

При проектировании микросервисов архитекторы смирились с дублированием реализации ради сохранения слабой связанности. Но как быть в случаях, когда сильная связанность *полезна*? В частности, для мониторинга, журналирования, аутентификации и авторизации, автоматических выключателей и множества других эксплуатационных возможностей, необходимых в каждом сервисе. Если каждая команда разработчиков сервиса сможет управлять этими зависимостями, возникнет хаос. Например, рассмотрим компанию PenultimateWidgets, которая пытается стандартизировать общее решение для мониторинга, чтобы упростить эксплуатацию своих сервисов. Если каждая команда отвечает за внедрение мониторинга для своего сервиса, как группа эксплуатации будет знать, что они это сделали? Кроме того, как быть, например, с унифицированием обновлений? Если необходимо обновить инструмент мониторинга во всей организации, как командам координировать этот процесс?

Довольно элегантное решение этой проблемы, появившееся в экосистеме микросервисов в последние несколько лет, — *паттерн Sidecar*, в основе которого лежит гораздо более ранний архитектурный паттерн, введенный Алистером Кокберном (Alistair Cockburn) и известный как *гексагональная архитектура*, см. рис. 5.17.

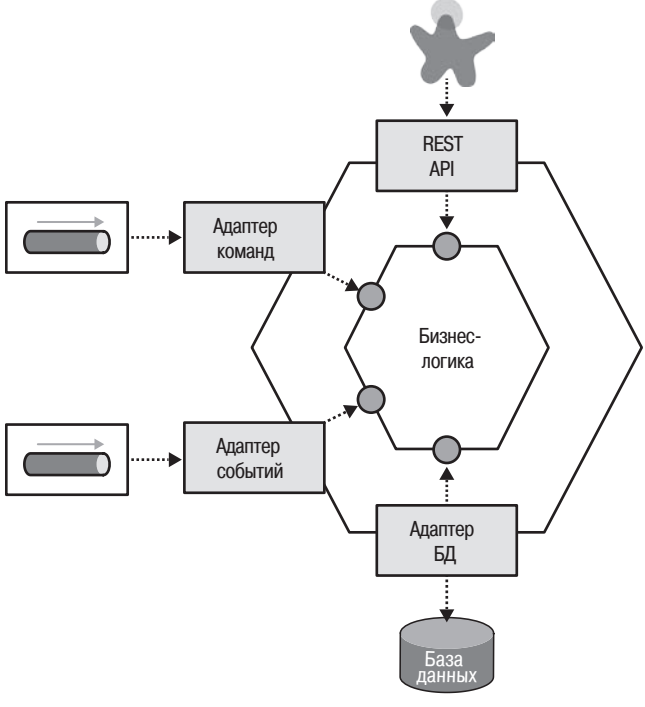


Рис. 5.17. Гексагональный паттерн отделяет логику предметной области от технической связанности

На рис. 5.17 то, что мы называем логикой предметной области, находится в центре шестиугольника, который окружен портами и адаптерами, связывающими его с другими частями экосистемы (на самом деле этот паттерн также известен как *Порты и адаптеры*). Хотя этот паттерн возник на несколько лет раньше микросервисов, он похож на современные микросервисы — с одним существенным отличием: верностью передачи данных. В гексагональной архитектуре база данных рассматривается как еще один подключаемый адаптер, но один из принципов DDD предполагает, что схемы данных и транзакционность должны находиться внутри структуры, такой как микросервисы.

Паттерн Sidecar использует тот же принцип, что и гексагональная архитектура, поскольку он отделяет логику предметной области от технической (инфраструктурной) логики. Например, рассмотрим два микросервиса на рис. 5.18.

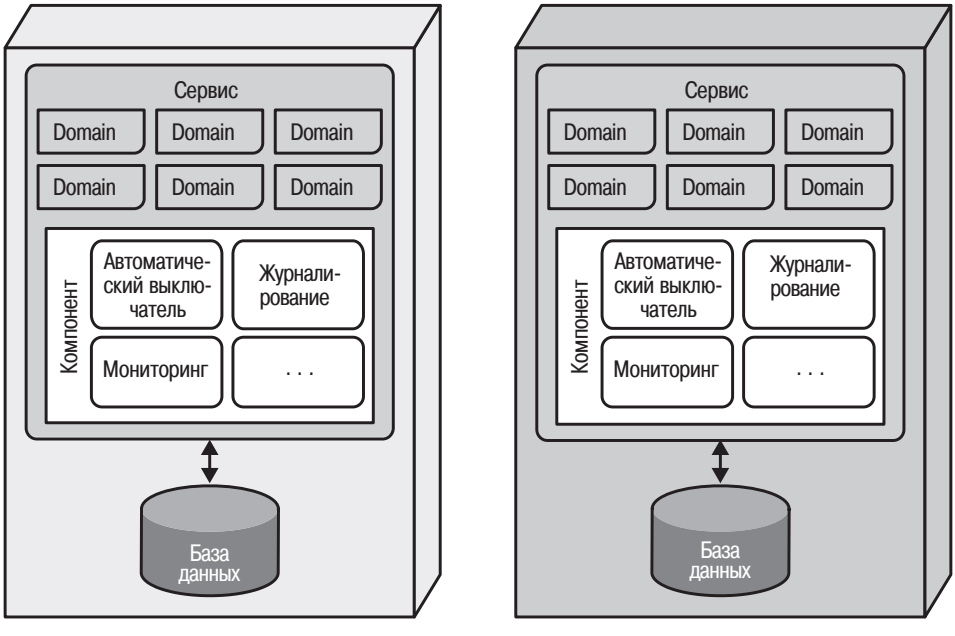


Рис. 5.18. Два микросервиса с одинаковыми эксплуатационными возможностями

На рис. 5.18 в каждом сервисе эксплуатационные задачи (более крупные блоки в нижней части сервиса) отделены от задач предметной области (прямоугольники с надписью Domain в верхней части сервиса). Если архитекторам необходимо обеспечить согласованность эксплуатационных задач, отделяемые части направляются в компонент sidecar, названный так по аналогии с коляской мотоцикла, реализация которого лежит либо на всех командах, либо управляется

централизованной инфраструктурной группой. Если каждый сервис будет содержать компоненты sidecar, эти компоненты сформируют согласованный эксплуатационный интерфейс, обычно подключаемый на уровне сервиса, как показано на рис. 5.19.

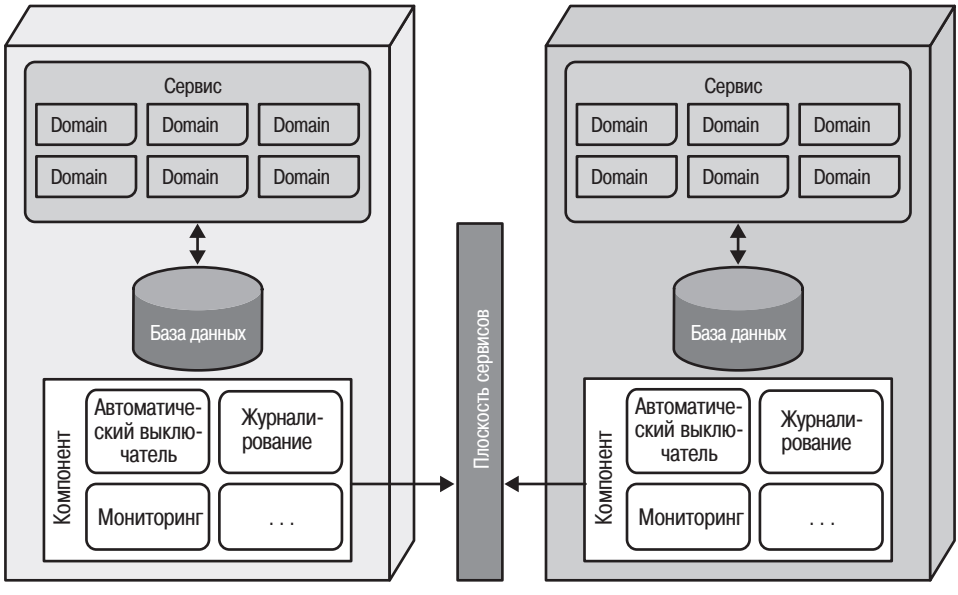


Рис. 5.19. Наличие общего компонента позволяет установить связи между микросервисами для последовательного управления

Если архитекторы и группа эксплуатации обеспечат безопасное добавление компонента sidecar (управляемого фитнес-функциями) в каждый сервис, эти сервисы составят сервисную сетку, показанную на рис. 5.20, на которой ячейки справа от каждого сервиса соединяются между собой, образуя «сетку».

Наличие сетки позволяет архитекторам и DevOps создавать дашборды, контролировать эксплуатационные характеристики, такие как масштаб, и реализовывать множество других возможностей.

Паттерн Sidecar позволяет группам управления, таким как архитекторы систем предприятий, эффективно сдерживать возникновение большого количества полиглотных сред: одним из преимуществ микросервисов является зависимость от интеграции, а не от общей платформы, что позволяет выбирать подходящий уровень сложности и возможностей для каждого сервиса. Однако по мере

увеличения количества платформ управлять ими централизованно становится все сложнее. Поэтому команды часто используют согласованность сервисной сетки для поддержки инфраструктуры и других общих свойств на разнородных платформах. К примеру, если необходимо объединить сервисы вокруг общего решения для мониторинга, без сервисной сетки придется создавать компоненты sidecar для каждой платформы, поддерживающей это решение.

Sidecar не только позволяет отделить эксплуатационные возможности от предметной области, но и выступает как паттерн *ортогонального повторного использования* для решения проблемы ортогональной связанности (см. врезку «Ортогональная связанность» на с. 150). Часто в архитектурных решениях требуются различные типы связанности, как в нашем примере связанности между предметной областью и эксплуатационными возможностями. Паттерн ортогонального повторного использования служит для повторного использования какого-либо компонента через один или несколько швов. Например, архитектуры микросервисов организованы вокруг предметных областей, но эксплуатационная связанность требует, чтобы эти области пересекались. Sidecar позволяет изолировать такие места в сквозном, но последовательном слое архитектуры.

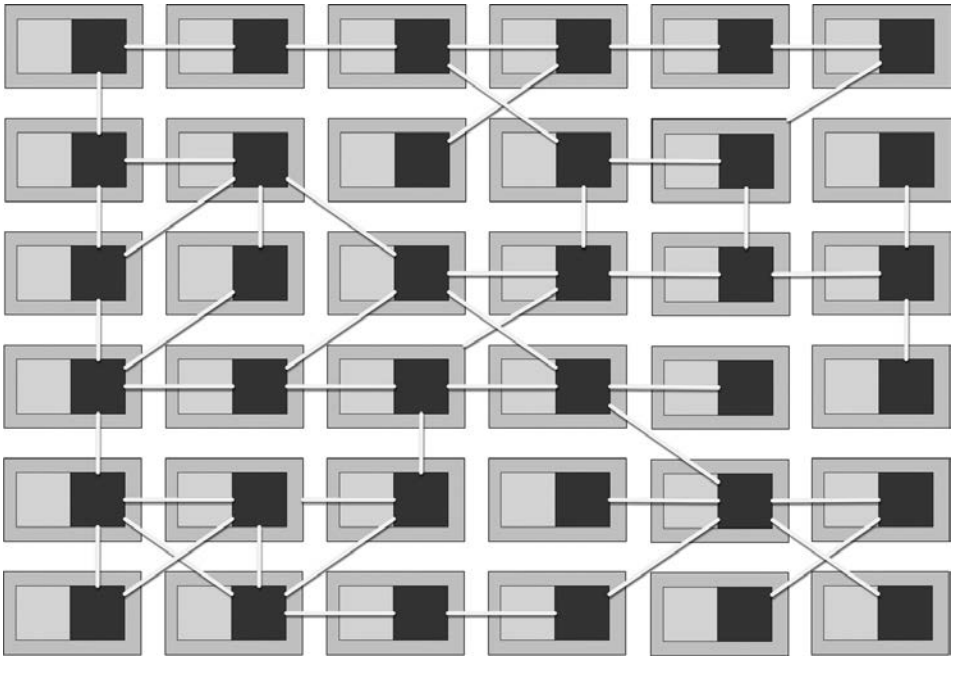


Рис. 5.20. Сервисная сетка — это набор эксплуатационных связей между сервисами

ОРТОГОНАЛЬНАЯ СВЯЗАННОСТЬ

В математике две прямые *ортогональны*, если они пересекаются под прямым углом, что также подразумевает, что они независимы. В архитектуре программного обеспечения две части архитектуры могут быть *ортогонально связаны*: два компонента с разным назначением, которые все же должны пересекаться, чтобы сформировать полноценное решение. Очевидный пример из этой главы касается эксплуатационных задач, таких как мониторинг, который необходим, но не зависит от поведения предметной области, например проверки каталога. Ортогональная связанность предоставляет точки пересечения, в которых задачи меньше всего переплетаются между собой.

Паттерн Sidecar и сервисная сетка — это наиболее чистые способы представления некоторых общих задач в распределенной архитектуре, и их можно использовать не только для эксплуатационной связанности (см. следующий раздел). Это архитектурный аналог паттерна Декоратор (Decorator) из книги «Design Patterns» группы Gang of Four («Банда четырех»)[1]: он позволяет архитектору «украсить» поведение в распределенной архитектуре, не затрагивая ее обычные связи.

Data Mesh: ортогональная связанность данных

Наблюдая за тенденциями в распределенных архитектурах, Жамак Дехгани (Zhamak Dehghani) и другие новаторы взяли основную идею ориентированного на предметную область разделения микросервисов — сервисную сетку — и паттерн Sidecar и применили их к аналитическим данным с небольшими изменениями. Как отмечалось в предыдущем разделе, паттерн Sidecar обеспечивает понятную организацию ортогональной связанности; разделение эксплуатационных и аналитических данных — еще один прекрасный пример именно такой связанности, но имеющей большую сложность, чем простая эксплуатационная связанность.

Определение сетки данных

Сетка данных (Data Mesh) воплощает децентрализованный подход к обмену, доступу и управлению аналитическими данными. Этот паттерн предназначен для решения широкого спектра аналитических задач, таких как составление отчетов, обучение моделей машинного обучения и аналитика. В отличие от предыдущего паттерна, он обеспечивает согласование архитектуры и прав владения данными

[1] *Джонсон Р., Влиссидес Дж., Хелм Р., Гамма Э.* «Приемы объектно-ориентированного проектирования. Паттерны проектирования». СПб., издательство «Питер».

с предметной областью бизнеса и предоставляет возможность однорангового потребления данных.

В основе Data Mesh лежат следующие принципы:

Данными владеет предметная область

Данные принадлежат тем предметным областям и совместно используются теми предметными областями, которые имеют к ним самое близкое отношение: являются либо их источником, либо их потребителями первого порядка. Архитектура обеспечивает распределенный обмен данными и доступ к ним из нескольких предметных областей одноранговым способом без промежуточных этапов преобразования, необходимых в хранилищах данных (data warehouses) или централизованном озере данных (data lake).

Данные как продукт

Для предотвращения силоса данных и стимулирования предметных областей к обмену данными Data Mesh вводит концепцию данных как продукта. Эта концепция определяет, какие роли и метрики успеха необходимы, чтобы предметные области предоставляли свои данные способом, обеспечивающим лучшее взаимодействие с потребителями данных в организации. Это приводит к введению нового архитектурного кванта — кванта продукта данных. Он позволяет поддерживать и предоставлять потребителям доступные, понятные, своевременные, безопасные и высококачественные данные. В этой главе рассматривается архитектурный аспект кванта продукта данных.

Самообслуживаемая платформа данных

Чтобы расширить возможности команд разработки предметных областей в части создания и поддержки своих продуктов данных, Data Mesh представляет новый набор возможностей самообслуживания. Эти возможности улучшают взаимодействие разработчиков и потребителей продуктов данных с платформой. Сюда входят такие функции, как декларативное создание продуктов данных, доступ к продуктам данных в сетке с помощью поиска и просмотра, а также управление появлением других интеллектуальных графов, таких как графы данных и знаний.

Вычислительное федеративное управление

Этот принцип гарантирует, что, несмотря на децентрализованное владение данными, общие требования управления, такие как требования соответствия, безопасности, конфиденциальности, качества данных и совместимости продуктов данных, будут последовательно соблюдаться во всех предметных областях. Data Mesh представляет федеративную модель принятия решений, включающую владельцев продуктов данных предметной области. Сформулированные ими политики автоматизированы и встроены в виде

кода в каждый продукт данных. Такой подход к управлению обуславливает добавление предоставляемого платформой компонента sidecar в каждый квант продукта данных для хранения и выполнения политик в точке доступа: чтения или записи данных.

Data Mesh — это обширная тема, которой посвящена книга «Data Mesh: Delivering Data-Driven Value at Scale»[1] (O'Reilly). В этой главе мы сосредоточимся на основном элементе этой архитектуры — *кванте продукта данных*.

Квант продукта данных

Основной принцип сетки данных лежит в основе современных распределенных архитектур, таких как микросервисы. По аналогии с *сервисной сеткой* команды создают *квант продукта данных* (data product quantum, DPQ), примыкающий к сервису, но связанный с ним, как показано на рис. 5.21.

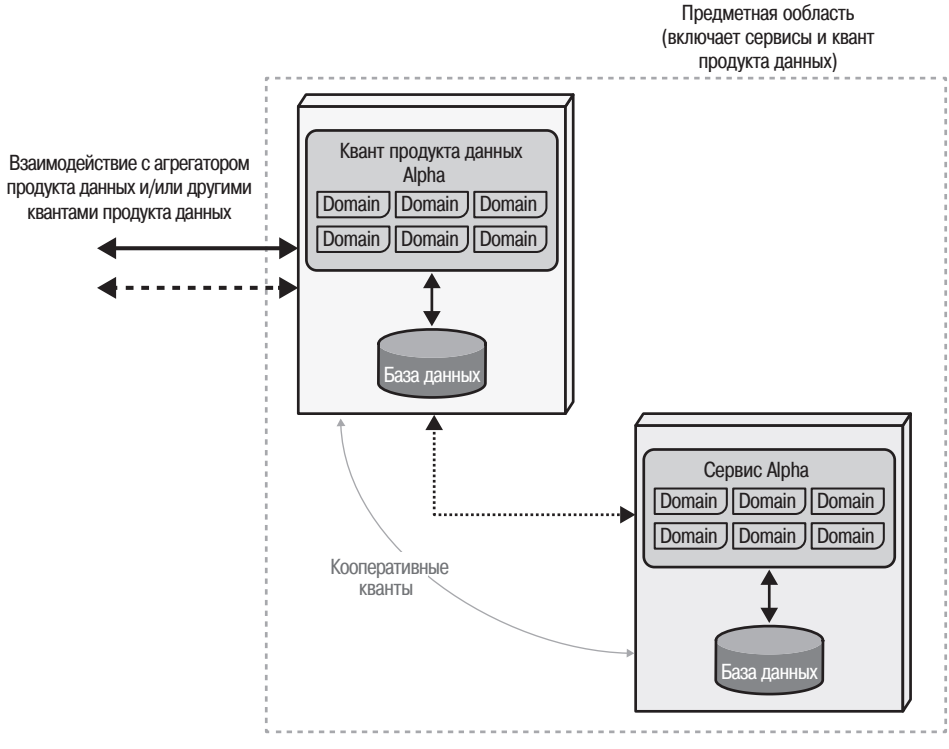


Рис. 5.21. Структура кванта продукта данных

[1] В 2024 году в издательстве «Питер» выходит книга Я. Майхшака, С. Балнояна, М. Сивиака «Data Mesh в действии», посвященная этой теме. — *Примеч. ред.*

Сервис *Alpha* содержит как поведенческие, так и транзакционные (вычислительные) данные. Предметная область также включает *квант продукта данных*, который содержит код и данные и выступает в роли интерфейса для общей части системы, отвечающей за аналитику и отчетность. DPQ действует как вычислительно независимый, но сильно сцепленный набор поведений и данных.

В современных архитектурах обычно существует несколько типов DPQ:

DPQ с привязкой к источнику (нативный) (Source-aligned (native))

Предоставляет аналитические данные от имени взаимодействующего кванта, обычно микросервиса, действующего как кооперативный квант.

Агрегирующий (Aggregate) DPQ

Агрегирует данные, полученные из нескольких источников, синхронно или асинхронно. Например, для получения некоторых агрегаций может быть достаточно асинхронного запроса; для других агрегатору DPQ требуется выполнить синхронные запросы DPQ с привязкой к источнику.

Целевой (Fit-for-purpose) DPQ

Специализированный DPQ для решения конкретной задачи, например составления аналитических отчетов, бизнес-аналитики, машинного обучения и др.

Предметная область может включать несколько DPQ в зависимости от характеристик архитектуры, необходимых для проведения анализа. Например, у разных DPQ могут быть разные требования к производительности.

Каждая предметная область, участвующая в анализе и бизнес-аналитике (business intelligence), содержит DPQ, как показано на рис. 5.22.

На рис. 5.22 DPQ представляет собой компонент, которым владеет предметная область, ответственная за реализацию сервиса. Он накладывается поверх информации, хранящейся в базе данных, и может асинхронно взаимодействовать с некоторым поведением предметной области. Квант продукта данных также обладает поведением и данными для целей анализа и бизнес-аналитики.

Каждый *квант продукта данных* является *кооперативным (cooperative) квантом* для сервиса.

Кооперативный квант

Вычислительно независимый квант, который взаимодействует со своим кооператором асинхронно и путем согласованности в конечном счете, но при этом сильно сцеплен с кооператором на условиях контракта и, как правило, более слабо — с квантом аналитики: сервисом, ответственным за предоставление отчетов, анализ, business intelligence и т. д.

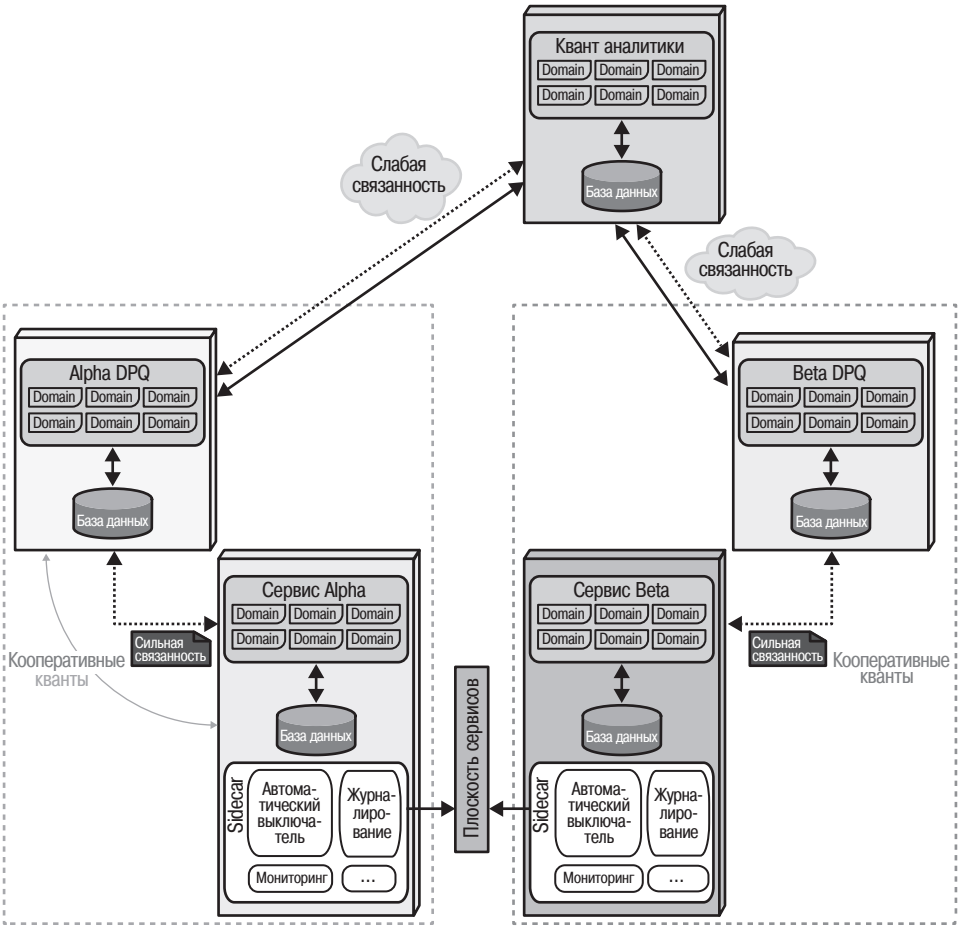


Рис. 5.22. Квант продукта данных — независимое, но сильно сцепленное дополнение к сервису

Хотя эти два взаимодействующих кванта эксплуатационно независимы, они представляют две стороны данных: сервис содержит эксплуатационные данные, а квант продукта данных — аналитические.

Определенная часть системы, ответственная за аналитику и business intelligence, формирует свою собственную предметную область и квант. Для работы этому кванту аналитики требуется статическая квантовая связанность с отдельными квантами продуктов данных, от которых он получает информацию. Этот сервис может осуществлять синхронные или асинхронные вызовы к DPQ, в зависимости от типа запроса. Например, некоторые DPQ имеют SQL-интерфейс для

аналитического DPQ, что позволяет выполнять синхронные запросы. Другие требования могут агрегировать информацию от нескольких DPQ.

Data Mesh — отличный пример инновационного гибрида архитектуры микросервисов и аналитических данных, а также «дорожная карта» для управления ортогональной связанностью в распределенных архитектурах. Концепции *sidecar* и *кооперативного кванта* позволяют архитекторам выборочно «накладывать» одну архитектуру поверх другой. Это позволяет проектировать системы на основе предметной области (например, согласно принципам DDD), предоставляя отдельным задачам управляемый доступ к необходимым ресурсам.

Итоги

Для архитекторов чрезвычайно важно понимать, какое влияние оказывает структура программной системы на ее способность к эволюции. Хотя существует множество архитектурных стилей, их основной характеристикой, определяющей способность создаваемой системы к эволюции, является контролируемая связанность. Идет ли речь о локализации коннасценции или ограниченном контексте в DDD, в любом случае, чтобы создавать эволюционирующие архитектуры, необходимо контролировать степень связанности реализации.

Благодаря контрактам части архитектуры могут взаимодействовать без создания жестких точек сцепления. Слабая связанность, гибкие контракты и контрактные фитнес-функции позволяют проектировать системы, которые отвечают требованиям и при этом просты в управлении или изменении.

ГЛАВА 6

Эволюционные данные

Современные программные проекты редко обходятся без реляционных и иных хранилищ данных, и со связанностью в базах данных часто работать еще сложнее, чем с архитектурной. Проектировщики баз данных, как правило, не привыкли проводить модульное тестирование или рефакторинг (хотя ситуация постепенно улучшается). Кроме того, базы данных часто служат точками интеграции, поэтому проектировщики неохотно вносят в них изменения из-за риска побочных эффектов.

Данные — важное измерение эволюционной архитектуры. Для создания архитектур, подобных микросервисам, требуется уделять гораздо больше внимания разделению данных, зависимостям, транзакционности и другим вопросам, которые раньше были вотчиной проектировщиков баз данных. Рассмотрение всех проблем проектирования эволюционных баз данных выходит за рамки этой книги. К счастью, наш соавтор Прамод Садаладж вместе со Скоттом Эмблером (Scott Ambler) написал книгу «Refactoring Databases», имеющую подзаголовок «Evolutionary Database Design»[1]. Мы рекомендуем читателям ознакомиться с ней, а здесь коснемся только тех вопросов, которые влияют на эволюционную архитектуру.

Проектирование эволюционных баз данных

Эволюционными можно считать базы данных, структуру которых можно менять по мере изменения требований. Схемы баз данных — это абстракции, подобные иерархии классов. Абстракции, создаваемые разработчиками и проектировщиками баз данных, должны отражать изменения реальных условий эксплуатации БД, в противном случае они постепенно перестанут соответствовать этим условиям.

[1] *Садаладж П., Эмблер С.* «Рефакторинг баз данных: эволюционное проектирование».

Эволюционные схемы

Как создавать системы, способные к эволюции, но при этом использующие традиционные инструменты, такие как реляционные базы данных? Чтобы база данных была способна эволюционировать, ее схема и код должны быть эволюционными. Вписать традиционное хранилище данных в непрерывный цикл обратной связи современных программных проектов позволяет непрерывная доставка. Изменяемые структуры баз данных необходимо проектировать так же, как исходный код: с помощью тестов, контроля версий и инкрементно.

Тестирование

Для обеспечения стабильности проектировщики баз данных и разработчики должны тщательно тестировать изменения в схемах баз данных. Если разработчики используют инструмент отображения данных, например инструмент объектно-реляционного отображения (object-relational mapper, ORM), им следует рассмотреть возможность добавления фитнес-функций, чтобы обеспечить синхронизацию отображений со схемами.

Контроль версий

Схемы баз данных необходимо версионировать вместе с кодом, который их использует. Исходный код и схемы баз данных взаимосвязаны — они не могут функционировать по отдельности. Их искусственное разделение не нужно и неэффективно.

Инкрементность

Изменения в схемах баз данных должны происходить так же, как в исходном коде: постепенно, по мере развития системы. Современная инженерная практика отходит от ручного обновления схем баз данных, предпочитая автоматизированные инструменты миграции.

Инструменты миграции баз данных — это утилиты, позволяющие разработчикам (или проектировщикам БД) вносить небольшие, инкрементные изменения в базу данных, которые автоматически применяются как часть пайплайна развертывания. Они очень разнообразны по своим возможностям — от простых инструментов командной строки до сложных прототипов IDE. Когда разработчикам необходимо внести изменения в схему, они пишут небольшие сценарии миграции базы данных (дельта-миграции), как показано в примере 6.1.

Пример 6.1. Простая миграция базы данных

```
CREATE TABLE customer (
       id BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1) PRIMARY KEY,
       firstname VARCHAR(60),
       lastname VARCHAR(60)
);
```

Инструмент миграции берет фрагмент SQL, показанный в примере 6.1, и автоматически применяет его к экземпляру БД разработчика. Если разработчику позже потребуется добавить дату рождения, то вместо изменения исходной миграции он может создать новую, которая изменит исходную структуру, как показано в примере 6.2.

Пример 6.2. Добавление даты рождения в существующую таблицу с помощью миграции

```
ALTER TABLE customer ADD COLUMN dateofbirth DATETIME;
```

После запуска миграции считаются неизменяемыми; изменять их — все равно что вести двойную бухгалтерию. Например, предположим, что разработчик Дэниэл выполнил миграцию в примере 6.2, и она стала 24-й по счету в проекте. Позже он понимает, что `dateofbirth` не нужна. Он может просто удалить 24-ю миграцию и, следовательно, колонку `dateofbirth`. Однако любой код, написанный после того, как Дэниэл выполнил миграцию, будет исходить из наличия колонки `dateofbirth` и перестанет работать, если понадобится сделать резервное копирование в промежуточную точку (например, чтобы исправить ошибку). Кроме того, в любой другой среде, где уже было применено это изменение, будет присутствовать удаленный столбец и возникнет несоответствие со схемой. Поэтому удалить столбец можно, создав новую миграцию.

В примере 6.2 разработчик изменяет существующую схему, добавляя новый столбец. Некоторые инструменты миграции также поддерживают возможность *отмены операции* (undo), как показано в примере 6.3. Она позволяет легко переходить между версиями схемы. Например, предположим, что версия проекта в репозитории исходного кода — 101 и необходимо вернуться к версии 95. Для исходного кода можно просто проверить версию 95 в системе контроля версий. Но как убедиться, что схема базы данных соответствует 95-й версии кода? Если использовать миграцию с возможностью *отмены*, можно вернуться назад к 95-й версии схемы, применяя по очереди каждую предыдущую миграцию до нужной версии.

Пример 6.3. Добавление даты рождения в существующую таблицу и отмена этой миграции

```
ALTER TABLE customer ADD COLUMN dateofbirth DATETIME;
--//@UNDO

ALTER TABLE customer DROP COLUMN dateofbirth;
```

Однако большинство команд не использует возможность *отмены* по трем причинам. Во-первых, если все миграции существуют, можно построить базу данных только до нужной точки, не откатываясь до предыдущей версии. В нашем

примере, чтобы восстановить версию 95, можно построить версии от 1 до 95. Во-вторых, зачем поддерживать две версии, прямую и обратную? Чтобы поддерживать возможность отмены, придется тестировать код, что иногда удваивает объем тестирования. В-третьих, создать полную отмену иногда чрезвычайно сложно. Представим, к примеру, что при миграции была удалена таблица — как в сценарии миграции сохранить все данные из нее в случае отмены удаления? Присвоить таблице префикс *DROPPED_* и сохранить ее? Но таблица постоянно будет изменяться, и вскоре данные в таблице *DROPPED* перестанут быть актуальными.

Миграции баз данных позволяют администраторам БД и разработчикам управлять изменениями схемы и кода инкрементно, как частями единого целого. Включение изменения базы данных в цикл обратной связи пайплайна развертывания обеспечивает дополнительные возможности для автоматизации и более ранней проверки в процессе сборки проекта.

Интеграция совместно используемых баз данных

Обратимся к распространенному шаблону интеграции — интеграции совместно используемых баз данных[1], когда база данных используется в качестве механизма обмена данными, как показано на рис. 6.1.

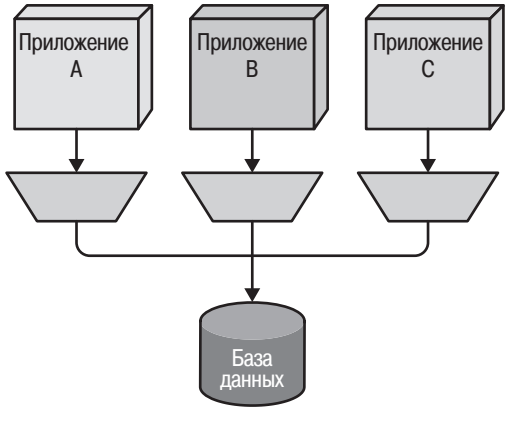


Рис. 6.1. Использование базы данных в качестве точки интеграции

[1] https://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html

На рис. 6.1 три приложения используют одну реляционную базу данных. Подобная схема интеграции часто по умолчанию применяется к проектам — в целях облегчения управления все проекты используют одну и ту же реляционную БД, так почему бы не сделать данные общедоступными для проектов? Однако архитекторы быстро обнаруживают, что при использовании базы данных в качестве точки интеграции схема БД во всех проектах не обновляется.

Что произойдет, если потребуется расширить возможности одного из связанных приложений и для этого изменить схему данных? Если приложение А внесет изменения в схему, это приведет к сбою двух других приложений. К счастью, как говорится в вышеупомянутой книге «Refactoring Databases», для распутывания такого рода связанности можно применить широко распространенный паттерн рефакторинга *Expand/Contract (Расширение/Сжатие)*. Многие методы рефакторинга баз данных позволяют избежать проблем со временем, встраивая в рефакторинг переходную фазу, как показано на рис. 6.2.

Этот паттерн использует начальное и конечное состояние, а во время перехода *старое* и *новое* состояния сохраняются. Благодаря наличию перехода обеспечивается обратная совместимость, и другие системы предприятия получают достаточно времени на изменение. В некоторых организациях переходное состояние может длиться от нескольких дней до нескольких месяцев.

Вот пример *Expand/Contract* в действии. Рассмотрим обычное эволюционное изменение — разделение столбца `name` (полное имя) на `firstname` (имя) и `lastname` (фамилия), что необходимо PenultimateWidgets в маркетинговых целях. У разработчиков есть начальное состояние, состояние расширения и конечное состояние, как показано на рис. 6.3.

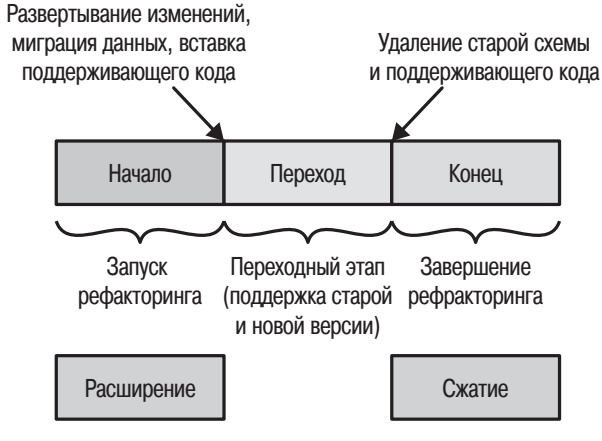


Рис. 6.2. Шаблон Expand/Contract для рефакторинга базы данных

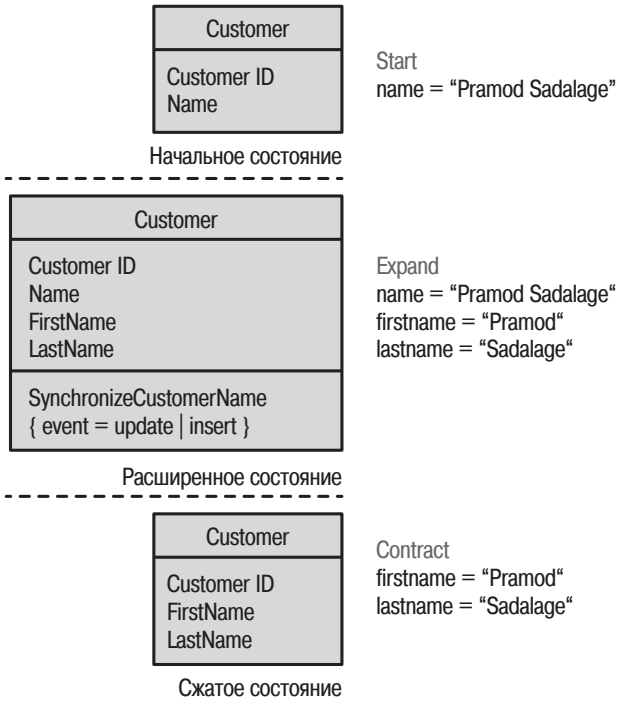


Рис. 6.3. Три состояния рефакторинга Expand/Contract

На рис. 6.3 полное имя отображается в виде одного столбца. Во время перехода разработчики PenultimateWidgets должны поддерживать обе версии, чтобы не нарушить возможные точки интеграции в базе данных. У них есть несколько вариантов, как разделить столбец `name` на `firstname` и `lastname`.

Вариант 1. Отсутствие точек интеграции и унаследованных данных

В этом случае разработчикам не нужно учитывать другие системы и управлять существующими данными, поэтому они могут добавить новые колонки и убрать старые, как показано в примере 6.4.

Пример 6.4. Простой случай без точек интеграции и унаследованных данных

```
ALTER TABLE customer ADD firstname VARCHAR2(60);
ALTER TABLE customer ADD lastname VARCHAR2(60);
ALTER TABLE customer DROP COLUMN name;
```

Для *варианта 1* рефакторинг прост: проектировщики БД вносят соответствующие изменения и продолжают обычную работу.

Вариант 2. Унаследованные данные, но без точек интеграции

Этот сценарий предполагает, что существующие данные требуется перенести в новые колонки, но внешних систем, которые используют эти данные, нет. В этом случае для переноса данных необходимо создать функцию, извлекающую переносимую информацию из существующего столбца, как показано в примере 6.5.

Пример 6.5. Унаследованные данные без интеграторов

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);
UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);
ALTER TABLE customer DROP COLUMN name;
```

Этот сценарий требует извлечения и переноса существующих данных, но в остальном он довольно прост.

Вариант 3. Существующие данные и точки интеграции

Это самый сложный и, к сожалению, самый распространенный сценарий. Компании необходимо перенести существующие данные в новые столбцы, при этом внешние системы связаны со столбцом `name`, который другие команды не могут перенести в нужные сроки, чтобы можно было использовать новые столбцы. Код SQL показан в примере 6.6.

Пример 6.6. Сложный случай с унаследованными данными и интеграторами

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);

UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);

CREATE OR REPLACE TRIGGER SynchronizeName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
  IF :NEW.Name IS NULL THEN
    :NEW.Name := :NEW.firstname||' '||:NEW.lastname;
  END IF;
  IF :NEW.name IS NOT NULL THEN
    :NEW.firstname := extractfirstname(:NEW.name);
    :NEW.lastname := extractlastname(:NEW.name);
  END IF;
END;
```

Чтобы построить этап перехода в примере 6.6, команда данных добавляет в базу данных триггер, который перемещает данные из старого столбца `name` в новые столбцы `firstname` и `lastname`, когда в базу данных добавляются данные из других систем. Таким образом новая система получает доступ к тем же данным. Аналогично столбцы `firstname` и `lastname` объединяют в столбец `name`, когда данные поступают из новой системы, чтобы другие системы имели доступ к корректно отформатированным данным.

После того как другие системы начнут использовать новую структуру (с отдельными столбцами для имен и фамилий), фаза *сжатия* завершается, а старый столбец удаляется:

```
ALTER TABLE Customer DROP COLUMN name;
```

Если данных много и удаление столбца займет много времени, можно задать для столбца статус «не используется» (если база данных поддерживает эту функцию):

```
ALTER TABLE Customer SET UNUSED name;
```

После удаления унаследованного столбца, если необходимо сохранить предыдущую версию схемы только для чтения, можно добавить функциональный столбец, чтобы сохранить доступ к базе данных для чтения:

```
ALTER TABLE CUSTOMER ADD (name AS
            (generatename (firstname,lastname)));
```

Как показано в каждом сценарии, разработчики и проектировщики могут использовать встроенные средства баз данных для создания эволюционирующих систем.

Expand/Contract — это разновидность паттерна Parallel Change (https://martinfowler.com/bliki/ParallelChange.html), широко используемого для безопасной реализации обратно-несовместимых изменений интерфейса.

Нежелательная запутанность данных

Данные и базы данных являются неотъемлемой частью большинства современных программных архитектур — разработчики, которые игнорируют этот ключевой аспект, терпят неудачу при попытке провести эволюцию своей архитектуры.

Базы данных и их проектировщики во многих организациях представляют собой головную боль, поскольку по тем или иным причинам их инструменты и инженерные практики устарели. Например, повседневные инструменты проектировщиков баз данных очень примитивны по сравнению с IDE любого разработчика.

Функции, привычные для разработчиков, такие как рефакторинг, тестирование вне контейнера, модульное тестирование, отслеживание зависимостей, линтинг, мокинг, заглушки и т. д., не знакомы проектировщикам БД.

ПРОЕКТИРОВЩИКИ БАЗ ДАННЫХ, ПОСТАВЩИКИ И ВЫБОР ИНСТРУМЕНТОВ

Почему инженерные практики сферы данных так сильно отстают от остальной разработки? Проектировщики БД во многом должны заниматься тем же, что и разработчики: тестированием, рефакторингом и т. д. Однако, в то время как инструменты для разработчиков постоянно развиваются, инновации в обработке данных идут гораздо более медленными темпами. При этом нельзя сказать, что инструменты просто отсутствуют — на сегодняшний день существует несколько полезных сторонних инженерных инструментов, способных повысить эффективность обработки данных. Но они плохо продаются. Почему?

Отношения поставщиков баз данных с потребителями довольно нетипичны. Например, проектировщики баз данных *DatabaseVendorX* необычайно преданы этому поставщику, потому что вся их дальнейшая работа, по крайней мере частично, будет зависеть от того, что они являются сертифицированными проектировщиками *DatabaseVendorX*, а не от того, какие задачи они выполняли до сих пор. Таким образом, у поставщиков БД в организациях по всему миру существуют своего рода тайные армии из сотрудников, преданных поставщику, а не своей компании. Проектировщики в такой ситуации игнорируют все сторонние инструменты и артефакты разработки. Результатом этого является застой инноваций в инженерной практике.

Для проектировщиков БД их поставщик — единственный светоч во вселенной, и их не волнует, что происходит в «черных дырах». Отставание в развитии инструментов обработки данных по сравнению с инструментами разработчиков — досадный побочный эффект такого отношения. В итоге разрыв между разработчиками и проектировщиками БД все увеличивается, поскольку они не используют общие методы проектирования. Полезно убедить проектировщиков БД принять практику непрерывной доставки, чтобы использовать новые инструменты и избавиться от зависимости от поставщика, чего они так стараются избежать.

К счастью, благодаря популярности БД с открытым исходным кодом и NoSQL гегемония поставщиков баз данных начала нарушаться.

Структуры данных в БД соединены с кодом приложений, и проектировщикам сложно проводить рефакторинг базы данных, не привлекая пользователей структур данных, таких как разработчики приложений, ETL-разработчики[1]

[1] ETL — от англ. Extract, Transform, Load — извлечение, преобразование и загрузка данных. — *Примеч. ред.*

и разработчики отчетов. Поскольку рефакторинг БД требует участия различных команд, координации ресурсов и установления приоритета со стороны команды продукта, провести его довольно сложно, и ему часто назначают низкий приоритет, что приводит к накоплению в БД неоптимальных структур и абстракций.

Транзакции с двухфазным коммитом

Когда архитекторы обсуждают связанность, разговор обычно сводится к классам, библиотекам и другим техническим вопросам. Однако в большинстве проектов существуют и другие возможности связанности, в том числе транзакции; это справедливо как для монолитных, так и для распределенных архитектур.

Транзакции — это особая форма связанности, поскольку традиционные инструменты, ориентированные на техническую архитектуру, не описывают транзакционное поведение. Центробежную и центростремительную связанность между классами легко определить с помощью различных инструментов. Гораздо труднее дело обстоит с контекстом транзакции. Подобно тому как связанность между схемами затрудняет эволюцию, то же самое делает и транзакционное сцепление, тесно связывающее составные части БД.

Транзакции в бизнес-системах формируются по целому ряду причин. Во-первых, бизнес-аналитикам нравится сама идея транзакции — операции, которая *фиксирует* определенный контекст, — независимо от технической сложности. Глобальная координация в сложных системах затруднена, и транзакции помогают ее осуществлять. Во-вторых, границы транзакций часто показывают, как на самом деле бизнес-концепции связаны друг с другом при их реализации. В-третьих, проектировщики баз данных могут владеть транзакционными контекстами, что затрудняет координацию разбиения данных на части со сцеплением, подобным связанности в технической архитектуре.

В главе 5 мы обсудили концепцию архитектурного кванта — наименьшей развертываемой архитектурной единицы, которая в отличие от традиционной связности охватывает зависимые компоненты, такие как базы данных. Связи в базах данных более жесткие, чем традиционные, из-за границ транзакций, определяющих, как работают бизнес-процессы. Архитекторы иногда строят системы с излишней гранулярностью, которая на самом деле не требуется бизнесу. Например, архитектуры микросервисов не очень хорошо подходят для транзакционных систем, потому что итоговый квант сервиса получается очень малым.

Архитекторы должны учитывать связанность во всех параметрах приложения: классах, пакетах/пространствах имен, библиотеках и фреймворках, схемах данных и транзакционных контекстах. Игнорирование любого из них (или их взаимодействий) затрудняет эволюцию архитектуры. В физике *сильное ядерное*

взаимодействие, связывающее частицы, является одним из самых сильных известных взаимодействий. Транзакционные контексты — это сильное ядерное взаимодействие квантов архитектуры.

Транзакции базы данных — аналог сильного ядерного взаимодействия, связывающий кванты архитектуры.

Хотя транзакций часто не избежать, необходимо стараться максимально ограничить транзакционные контексты, поскольку они образуют тесные узлы, не позволяющие менять одни компоненты или сервисы, не затрагивая другие. Что еще более важно, при планировании изменений в архитектуре необходимо учитывать такие особенности, как транзакционные границы.

Как мы обсудим в главе 9, переход от монолитного архитектурного стиля к гранулированному следует начинать с создания небольшого числа более крупных сервисов. При создании новой архитектуры микросервисов следует тщательно ограничивать размер контекстов сервисов и данных. Однако не стоит воспринимать термин «*микросервисы*» слишком буквально — каждый сервис не обязательно должен быть маленьким; скорее он должен включать полезный ограниченный контекст.

При реструктуризации существующей схемы базы данных часто бывает трудно добиться подходящей гранулярности. Многие проектировщики баз данных тратят десятилетия на создание схемы базы данных и не желают выполнять обратную операцию. Часто наименьшая допустимая гранулярность сервисов определяется необходимыми для поддержки бизнеса транзакционными контекстами. Стремление к более подробному уровню гранулярности может привести лишь к образованию излишней связанности, если желаемая гранулярность будет противоречить структуре данных. Создание архитектуры, которая структурно противоречит задачам разработчиков, — это плохой вариант метаработы, описанный в разделе «Миграция архитектур» на с. 187 (глава 7).

Возраст и качество данных

Еще одна проблема, характерная для крупных компаний, — это фетишизация данных и баз данных. Мы не раз слышали от технических директоров: «Приложения не важны, потому что они долго не живут, *главная ценность — это схемы данных, потому что они вечны!*» Схемы действительно меняются реже, чем код, но они все равно представляют собой абстракцию реального мира. Хотя это и неудобно, реальные условия со временем изменяются. Проектировщики

баз данных, которые считают, что схемы никогда не меняются, игнорируют реальность.

Но если проектировщики никогда не проводят рефакторинг БД для внесения изменений в схему, то как они вносят изменения, чтобы внедрить новые абстракции? К сожалению, чаще всего — просто *добавляя еще одну таблицу*. Вместо того чтобы вносить изменения в схему, рискуя сломать существующие системы, они просто присоединяют новую таблицу к исходной с помощью примитивов реляционной БД. Хотя в краткосрочной перспективе это работает, такой подход затемняет лежащую в основе реальную абстракцию: в реальности одна сущность представлена несколькими явлениями. Это приводит к тому, что проектировщики БД, которые редко действительно реструктурируют схемы, создают окаменевшие системы с причудливыми группировками и объединениями. Если команда не реструктурирует базу данных, она не сохраняет ценный ресурс предприятия и оставляет после себя застывшие останки каждой версии схемы, наложенные друг на друга с помощью присоединения таблиц.

Еще одна огромная проблема — качество унаследованных данных. Часто данные переживают не одно поколение ПО, в каждом из которых были свои особенности сохранения, в результате чего эти данные в лучшем случае непоследовательны, а в худшем представляют собой мусор. Желание сохранять каждый бит данных привязывает архитектуру к прошлому, заставляя придумывать сложные обходные пути, чтобы все продолжало работать.

Прежде чем строить эволюционную архитектуру, убедитесь, что данные тоже смогут эволюционировать как в плане схемы, так и в плане качества. Плохая структура требует рефакторинга, а проектировщики баз данных должны делать все необходимое для обеспечения качества данных. Лучше устранять эти проблемы на ранней стадии, а не создавать для их решения сложные механизмы, которые необходимо постоянно поддерживать.

Унаследованные схемы и данные имеют ценность, но они также ограничивают способность к эволюции. Архитекторы, проектировщики баз данных и представители бизнеса должны совместно определить, что представляет *ценность* для организации — постоянное хранение унаследованных данных или возможность эволюционных изменений. Выделите действительно ценные данные и сохраняйте их, а устаревшие данные сделайте доступными для справки, чтобы от них не зависела эволюция архитектуры.

Отказ от рефакторинга схем или удаления старых данных привязывает архитектуру к прошлому, рефакторинг которого провести нельзя.

Практический пример: эволюция маршрутизации в PenultimateWidgets

PenultimateWidgets решила реализовать новую схему маршрутизации между страницами с помощью паттерна Хлебные крошки (breadcrumbs). Для этого необходимо изменить способ маршрутизации между страницами (с использованием собственного фреймворка). Страницам, реализующим новый механизм маршрутизации, требуется больше контекста (исходная страница, состояние рабочего процесса и т. д.) и, следовательно, больше данных.

Квантом сервиса маршрутизации PenultimateWidgets сейчас служит одна таблица обработки маршрутов. Для новой версии требуется больше информации, поэтому структура таблицы будет более сложной. Рассмотрим начальную точку, показанную на рис. 6.4.

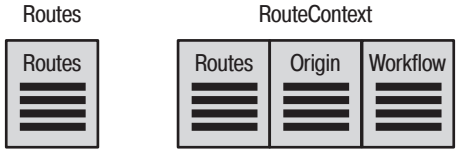


Рис. 6.4. Начальная точка внедрения новой маршрутизации

Не все страницы в PenultimateWidgets будут реализовывать новую маршрутизацию одновременно, поскольку бизнес-подразделения работают с разной скоростью. Поэтому сервис маршрутизации должен поддерживать как старую, так и новую версию. Мы узнаем, как это обеспечить, в главе 7. В данном случае тот же сценарий необходимо реализовать на уровне данных.

Используя паттерн Expand/Contract, можно создать новую структуру маршрутизации и сделать ее доступной через вызов сервиса. Внутри обе таблицы маршрутизации имеют триггер, ассоциированный с колонкой `route`, так что изменения в одной из них автоматически реплицируются в другую, как показано на рис. 6.5.

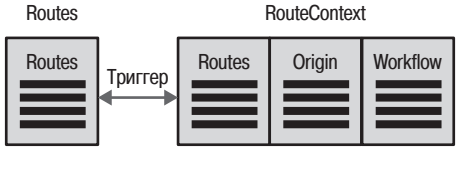


Рис. 6.5. Состояние перехода, в котором сервис поддерживает обе версии маршрутизации

Как показано на рис. 6.5, сервис будет поддерживать оба API до тех пор, пока старая версия маршрутизации необходима. По сути, приложение теперь поддерживает две версии данных маршрутизации.

Когда необходимость в старом сервисе отпадает, старую таблицу и триггер можно удалить, как показано на рис. 6.6.

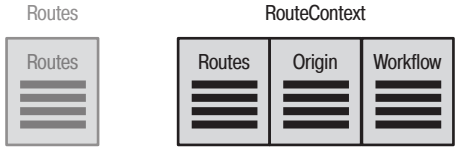


Рис. 6.6. Конечное состояние таблиц маршрутизации

На рис. 6.6 все сервисы перешли на новую версию маршрутизации, что позволяет удалить старую. Рабочий процесс такой же, как на рис. 6.2.

База данных может развиваться параллельно с архитектурой, если разработчики применяют надлежащие инженерные практики, такие как непрерывная интеграция, управение версиями и т. д. Возможность легко менять схему базы данных очень важна: база данных представляет собой абстракцию, основанную на реальных условиях, которые могут неожиданно измениться. Хотя сопротивление изменениям в абстракциях данных выше, чем у поведения, абстракции все равно должны эволюционировать. При построении эволюционной архитектуры вопросы данных должны рассматриваться в первую очередь.

Рефакторинг баз данных — это важный практический навык для проектировщиков БД и разработчиков. Данные — основа многих приложений. Для создания эволюционирующих систем необходимо использовать эффективные методы работы с данными и другие современные инженерные практики.

От натива к фитнес-функции

Иногда выбор программной архитектуры приводит к возникновению проблем в других частях экосистемы. Архитектура микросервисов, которая предполагает наличие своей базы данных для каждого ограниченного контекста, изменила традиционный взгляд проектировщиков на базы данных: они привыкли к ис-

пользованию одной реляционной БД и удобствам этой модели. К примеру, они уделяют пристальное внимание ссылочной целостности, чтобы обеспечить корректность связей в структуре данных.

Но что, если архитекторам необходимо разбить базы данных на части, обслуживающие отдельные сервисы, такие как микросервисы, — как им убедить скептически настроенных проектировщиков БД, что преимущества микросервисов перевешивают отказ от некоторых проверенных механизмов?

Поскольку это одна из форм управления, можно успокоить проектировщиков, включив в сборку непрерывные фитнес-функции, чтобы обеспечить поддержку целостности важных частей и решить другие проблемы.

Ссылочная целостность

Ссылочная целостность — это форма управления скорее на уровне схемы данных, а не связанности архитектуры. Однако для архитектора и то и другое влияет на возможность эволюции приложения из-за увеличения связанности. Например, зачастую из-за ссылочной целостности проектировщики баз данных, хоть и неохотно, разбивают таблицы на отдельные базы данных, но из-за связанности сервисы, связанные с БД, не могут изменяться.

Ссылочная целостность в базах данных касается первичных ключей и их связей. В распределенных архитектурах существуют уникальные идентификаторы для объектов, часто выраженные в виде GUID или другой случайной последовательности. Таким образом, архитекторы должны создать фитнес-функции, чтобы гарантировать, что если какой-то элемент будет удален владельцем информации, об этом узнают другие сервисы, которые все еще могут ссылаться на удаленную сущность. Для решения этой задачи существует целый ряд паттернов в событийно-ориентированной архитектуре; один из них показан на рис. 6.7.

На рис. 6.7, когда пользовательский интерфейс отклоняет сделку через сервис `Trader Blotter`, он распространяет сообщение в долговременной очереди сообщений, которую отслеживают все заинтересованные сервисы, обновляя или удаляя изменения по мере необходимости.

Хотя ссылочная целостность в базах данных — это мощный инструмент, она иногда создает нежелательную связанность, которую необходимо оценивать относительно преимуществ.

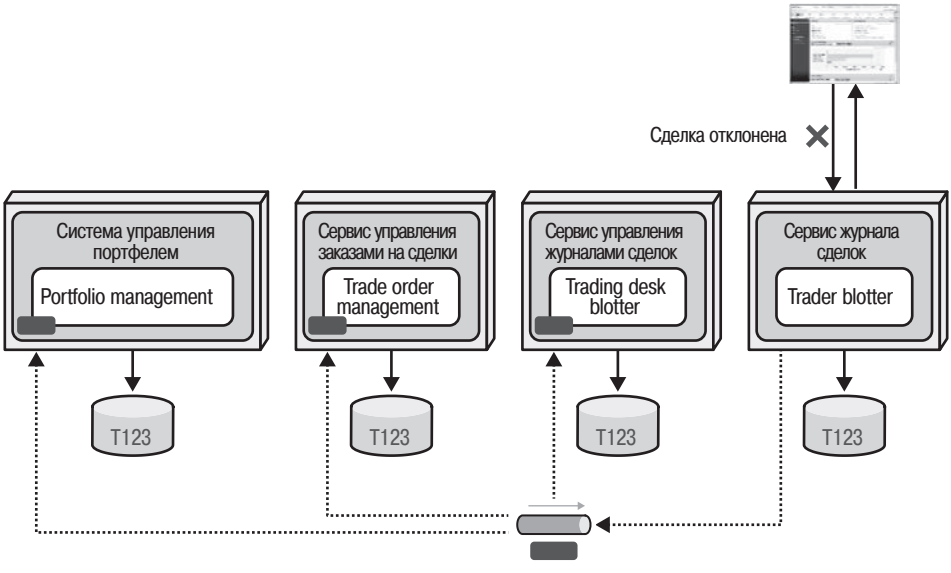


Рис. 6.7. Использование синхронизации данных на основе событий для обработки ссылочной целостности

Дублирование данных

Используя единую реляционную базу данных, проектировщики обычно уже не рассматривают две операции — *чтение* и *запись* — как отдельные. Однако в архитектуре микросервисов необходимо более тщательно продумывать, какие сервисы могут обновлять информацию, а какие — только читать. На рис. 6.8 показан распространенный сценарий, с которым сталкиваются многие команды, только начинающие работать с микросервисами.

Нескольким сервисам необходим доступ к ключевым частям системы, таким как справочные таблицы, таблицы аудита, конфигурации и данных клиентов. Как его обеспечить? Решение, показанное на рис. 6.8, предоставляет всем заинтересованным сервисам доступ к таблицам, что удобно, но нарушает один из принципов архитектуры микросервисов — избегать привязки сервисов к общей базе данных. Если схема любой из этих таблиц изменится, это отразится на связанных с таблицей сервисах и, возможно, потребует их изменения.

Альтернативный подход представлен на рис. 6.9.

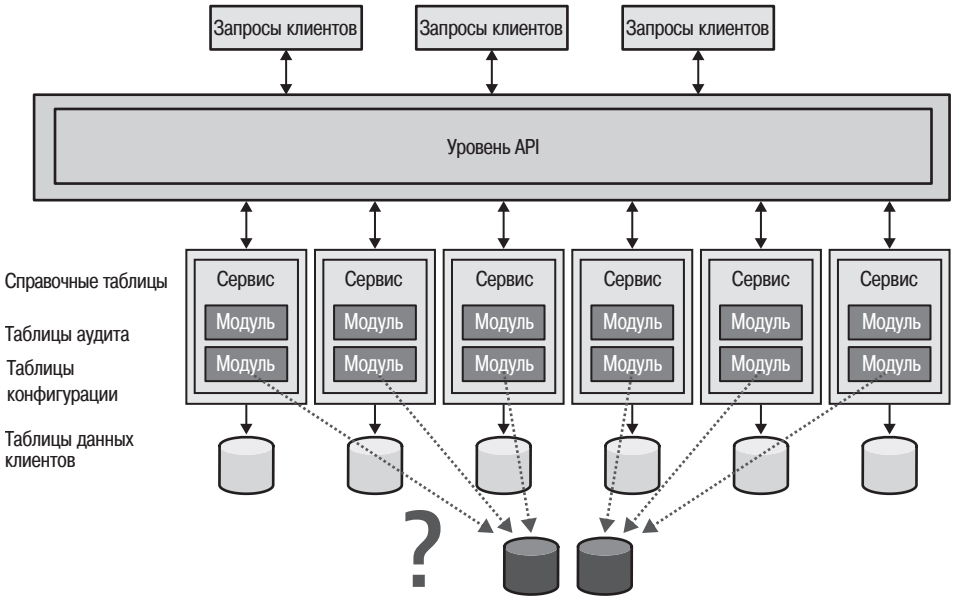


Рис. 6.8. Управление совместно используемой информацией в распределенной архитектуре

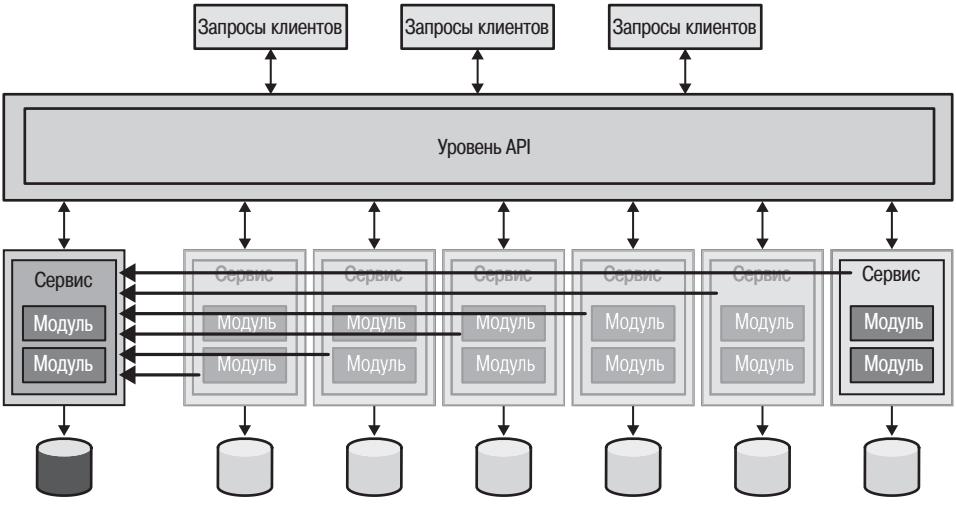


Рис. 6.9. Модель совместно используемой информации как услуги

На рис. 6.9, согласно принципу архитектуры микросервисов, каждый совместно используемый блок информации смоделирован как отдельный сервис. Однако это обнажает одну из проблем микросервисов — большой объем межсервисного взаимодействия, которое влияет на производительность.

Важно тщательно проанализировать, кто должен *владеть* данными (то есть кто может их обновлять), а кто может только *читать* некоторые версии данных. В решении, показанном на рис. 6.10, доступ для чтения организован с использованием внутрипроцессного кэширования.

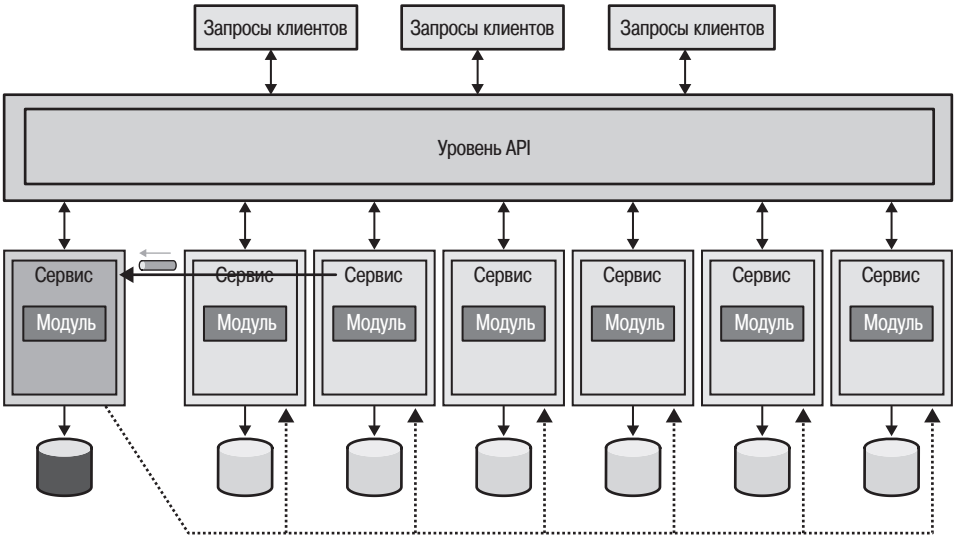


Рис. 6.10. Использование кэширования для доступа только для чтения

На рис. 6.10 сервис слева «владеет» данными. Однако при запуске каждый заинтересованный сервис считывает и кэширует необходимые ему данные с соответствующей частотой обновления кэшированной информации. Если одному из сервисов справа необходимо обновить совместно используемую информацию, он делает это через запрос к сервису-владельцу, который затем публикует изменения.

В современных архитектурах существуют различные стратегии управления доступом к данным и их обновлением. Примеры включают контроль версий, масштабируемость управления соединениями, отказоустойчивость, архитектурные кванты, оптимизацию типов баз данных, транзакции баз данных и отношения данных и подробно рассматриваются в книге «Software Architecture: The Hard Parts»[1].

[1] *Форд Н.* и др. «Современный подход к программной архитектуре: сложные компромиссы». СПб., издательство «Питер».

Замена триггеров и хранимых процедур

Другим механизмом, который часто используют проектировщики баз данных, являются хранимые процедуры на родном языке баз данных — SQL. Хотя это мощный и эффективный вариант манипулирования данными, у него есть и недостатки, важные в современных условиях. Например, хранимые процедуры трудно поддаются модульному тестированию, часто имеют слабую поддержку рефакторинга и разделяют модели поведения в исходном коде.

При переходе на микросервисы проектировщикам БД часто приходится проводить рефакторинг хранимых процедур, поскольку данные больше не хранятся в одной базе. В этом случае требуется перенести поведение в код, определив, какой объем данных и как переносить. В современных базах данных NoSQL можно вводить триггеры или бессерверные функции, которые запускаются, если в данных происходят изменения. Рефакторинг требуется во всем коде БД.

Для извлечения поведения хранимых процедур в код приложения архитекторы могут использовать тот же паттерн Expand/Contract, а именно его разновидность Migrate Method from Database[1] (перенос метода из базы данных), как показано на рис. 6.11.

На этапе расширения разработчики добавляют метод замены в сервис *администрирования виджетов* и проводят рефакторинг других сервисов, чтобы те обращались к сервису *администрирования виджетов*. Пока команда не заменит функциональность в хорошо протестированном коде, новый метод действует как сквозной для хранимой процедуры. В этот период приложение поддерживает вызовы как сервиса, так и хранимой процедуры. На этапе *сжатия* архитекторы могут использовать фитнес-функцию, чтобы убедиться, что все зависимости обращаются к сервису, и удаляют хранимую процедуру. Это вариант паттерна Strangler Fig (Фикус-душитель)[2] для баз данных.

Другим вариантом может быть отказ от рефакторинга хранимой процедуры и создание более широкого контекста данных, как показано на рис. 6.12.

На рис. 6.12 вместо того, чтобы заменять триггеры и хранимые процедуры кодом, было решено сделать сервис более гранулярным. Универсального решения быть не может; необходимо оценивать компромиссы в каждом конкретном случае.

[1] https://www.databaserefactoring.com/MigrateMethodFromDatabase.html

[2] https://martinfowler.com/bliki/StranglerFigApplication.html

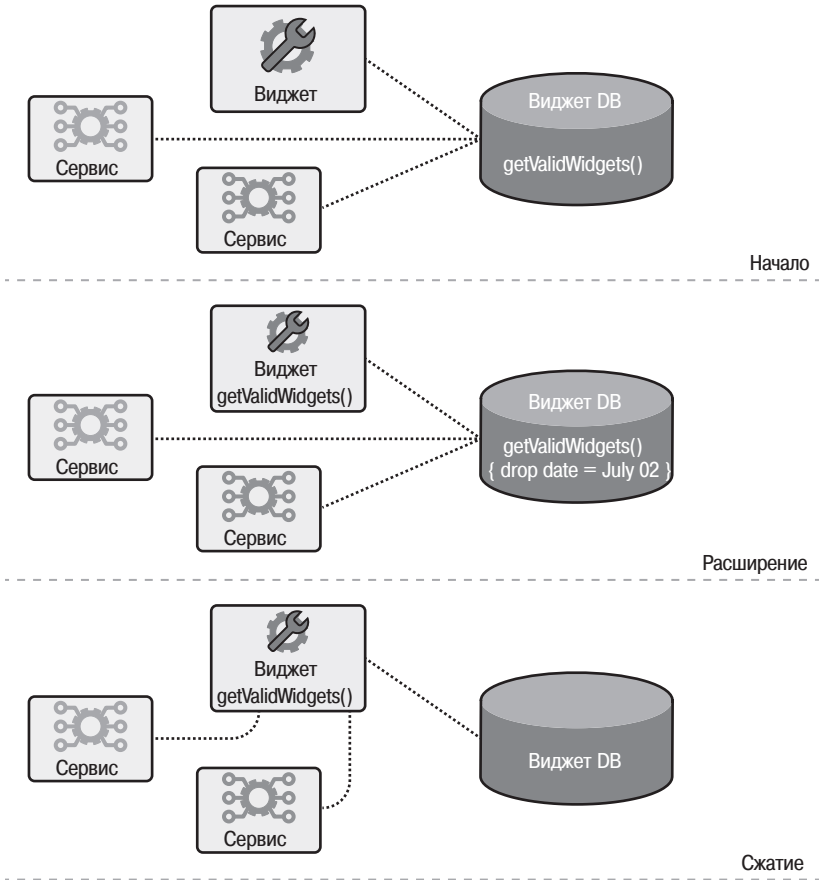


Рис. 6.11. Извлечение кода базы данных в сервисы

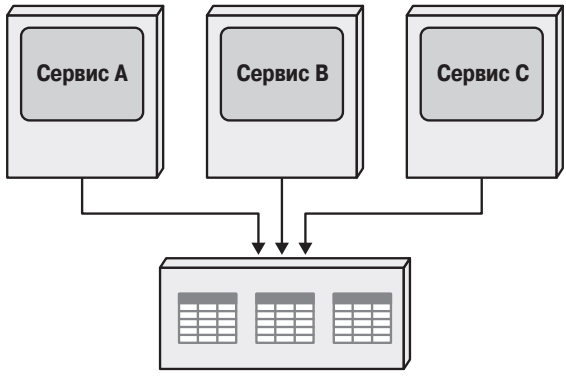


Рис. 6.12. Построение более широкого контекста данных и сохранение хранимых процедур

Практический пример: эволюция реляционной модели в нереляционную

Многие компании, такие как PenultimateWidgets, начинают с монолитных приложений, и на то есть веские стратегические причины: время выхода на рынок, простота, неопределенность рынка и др. В таких приложениях обычно всего одна реляционная база данных, что десятилетиями было отраслевым стандартом.

Разрушение монолита — дополнительная возможность преобразовать персистентность. Например, для каталогизации и категоризации аналитических данных лучше использовать базу данных графов. Для некоторых предметных областей лучше использовать базы данных пар имя/значение. Одно из преимуществ высокораспределенных архитектур, таких как микросервисы, заключается в возможности подбирать механизмы сохранения, основываясь на проблеме, а не на произвольном стандарте. Переход от монолита к микросервисам может выглядеть как на рис. 6.13.

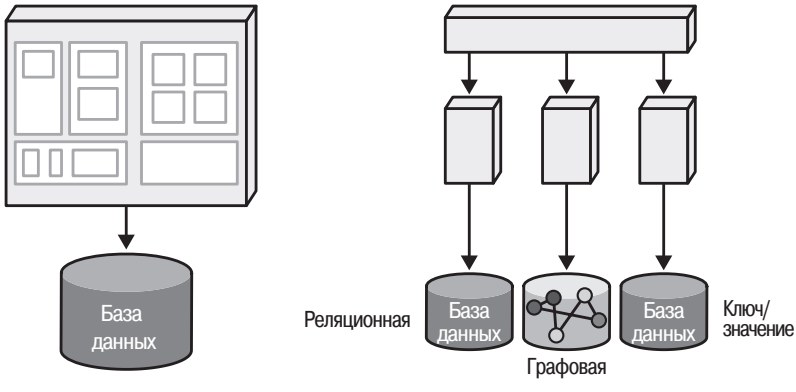


Рис. 6.13. Переход PenultimateWidgets от монолита к архитектуре микросервисов

На рис. 6.13 каталог, аналитические данные (используемые для составления прогнозов рынка и другой бизнес-аналитики) и операционные данные (такие как состояние продаж, транзакции и т. д.) находятся в одной базе данных, иногда приспосабливаясь к различным применениям. Однако при переходе к микросервисам появляется возможность разбить монолитные данные на несколько более репрезентативных типов. Например, некоторые данные могут лучше подходить для хранения в качестве пар ключ/значение, а не для строго реляционных баз данных. Точно так же на решение проблем, с которыми с помощью базы данных графов можно справиться за несколько секунд, в реляционной базе данных могут уйти часы или дни.

Однако переход от единой базы данных к нескольким (даже одного типа) может вызвать проблемы; программная архитектура целиком построена на компромиссах. Архитекторам может быть трудно убедить проектировщиков БД разбить персистентность на несколько хранилищ данных, поэтому им следует обращать их внимание на компромиссы, предлагаемые каждым подходом.

Итоги

Последняя часть нашего определения эволюционной архитектуры касается *множества измерений*, и именно данные являются основной неархитектурной проблемой, которая влияет на эволюцию программных систем. Появление современных распределенных архитектур, таких как микросервисы, привело к тому, что вопросы, которые раньше лежали в сфере ответственности исключительно проектировщиков баз данных, теперь легли на плечи архитекторов. Переопределение структуры вокруг ограниченных контекстов означает разделение данных, что означает принятие целого ряда компромиссов.

Архитекторы должны тщательно продумывать влияние данных на архитектуру и работать совместно с проектировщиками баз данных и с разработчиками систем.

ЧАСТЬ III

Влияние

Хотя мы рассматривали *механику* и *структуру* по отдельности, в реальных системах они взаимосвязаны. Часть III этой книги описывает, как пересекаются практики разработки, речь о которых шла в части I, и особенности структуры, рассмотренные в части II.

ГЛАВА 7

Создание эволюционных архитектур

До сих пор мы рассматривали две основные составляющие эволюционной архитектуры — механику и структуру — по отдельности. Теперь у нас есть достаточно контекста, чтобы связать их вместе.

Многие из обсуждаемых здесь концепций не новы — это скорее старые идеи, рассмотренные через новую призму. Например, тестирование существует уже много лет, но не с таким акцентом на проверку архитектуры, как фитнес-функция. *Непрерывная доставка* определила идею пайплайнов развертывания. Эволюционная архитектура стремится сделать автоматизацию управляемой.

Многие организации применяют непрерывную доставку в целях повышения эффективности разработки, что само по себе достойно уважения. Однако мы идем дальше и используем ее возможности для создания более сложных вещей — архитектуры, которая эволюционирут вместе с реальным миром.

Как же использовать преимущества этих методов в проектах, как существующих, так и новых?

Принципы эволюционной архитектуры

Как механику, так и структуру в эволюционной архитектуре определяют пять общих принципов. Рассмотрим их подробнее.

Последний ответственный момент

Мир agile-разработки уже давно оценил достоинства *последнего ответственного момента* (last responsible moment): откладывать принятие решений насколько это возможно долго, но не дольше. Слишком раннее принятие решений ведет к усложнению систем, а слишком позднее — к тому, что цели архитектуры не будут достигнуты.

Цель этого подхода заключается не в максимальном увеличении задержки. Если в процессе принятия решения найти правильную переломную точку, мы получим максимальное количество информации. Это полезно, потому что в конечном счете работа архитектора заключается в анализе компромиссов, и чем больше у него информации, тем больше критериев он сможет проанализировать.

Принимая решения слишком рано, архитекторы стремятся сохранить открытость, выбирая более общие решения. Однако это может привести к усложнению конкретных реализаций, что нивелирует преимущества обобщения для разработчиков.

Заранее определите цели и соответственно расставьте приоритеты в принятии решений.

Эволюционная архитектура и эволюционная разработка

Проблема обеспечения способности архитектуры к *эволюции* должна стать первоочередной для архитекторов. Это подразумевает использование объективных измерений при анализе характеристик архитектуры, а также выбор оптимальной степени связанности и решений, позволяющих избежать хрупкости в архитектуре.

Как мы обсуждали в главе 6, при проектировании архитекторы должны в первую очередь учитывать данные и другие внешние точки интеграции (статическая связанность для кванта архитектуры). Например, проектировщики баз данных должны непрерывно интегрировать изменения базы данных, как и код, а архитекторы должны рассматривать зависимости данных наравне с зависимостями кода.

Этот принцип применим не только к целостным частям архитектуры, но и ко всему процессу разработки и подбору инструментов. Чтобы обеспечить наименьшее трение и максимально информативную обратную связь, обращайте внимание на оба фактора.

Закон Постела

> Консервативно относитесь к своей деятельности и либерально — ко вкладам других.
>
> — *Джон Постел (Jon Postel)*

Важным принципом, дополняющим обсуждение контрактов в разделе «Контракты» на с. 134 (глава 5), является *закон Постела,* который направлен на максимальное сглаживание связанности. Применительнно к контрактам и взаимодействиям он полезен для обеспечения эволюционности:

Будьте консервативны в том, что вы отправляете

Не отправляйте больше информации, чем необходимо, — если сервис запрашивает только номер телефона, не отправляйте крупную структуру данных. Чем больше информации в контракте, тем чаще другие точки сцепления будут использовать ее, ужесточая контракт, который в противном случае мог быть более свободным.

Будьте либеральны в том, что вы принимаете от других

Вы можете принять больше информации, чем потребляете. Вам не нужно потреблять больше информации, чем необходимо, даже если она у вас есть. Если вам нужен только номер телефона, не стройте протокол для всего адреса, проверьте только номер телефона. Это отделяет сервис от информации и точек сцепления, которые ему не нужны.

Используйте версионирование при разрыве контракта

Архитекторы должны соблюдать контракты в интеграционной архитектуре (автоматизированной с помощью контрактов, управляемых потребителями), что означает внимание к эволюции функциональности сервисов.

О законе Постела в контексте архитектуры написано много, и не зря — он предлагает хорошее решение вопроса разделения, а оно, в свою очередь, помогает создавать эволюционную архитектуру.

Архитектура должна быть тестируемой

Многие архитекторы сетуют, что в их архитектуре есть места, сложные для тестирования, что неудивительно, поскольку при проектировании тестируемость обычно не ставится в приоритет. И наоборот, если архитекторы проектируют свою архитектуру с учетом тестирования, они упрощают тестирование ее отдельных частей. Например, в экосистеме микросервисов существует

множество исследований и инструментов, облегчающих тестирование, что повышает ее общую эволюционность. В целом существует корреляция между системой, которую трудно тестировать, и системой, которую трудно поддерживать и улучшать.

В хорошей тестируемой архитектуре также соблюдается *принцип единственной ответственности* (single responsibility principle): каждая часть системы должна иметь одну ответственность. Например, прежде был распространен антипаттерн смешения бизнес-логики с инфраструктурой обмена сообщениями с помощью таких инструментов, как сервисная шина предприятия (Enterprise Service Bus). Мы поняли, что смешение ответственности затрудняет тестирование любого отдельного поведения.

Закон Конвея

Точки сцепления непредсказуемым образом возникают в порой неожиданных местах. Структура команды и ее влияние на архитектуру — это ключ к эволюционной архитектуре; подробнее о законе Конвея см. в разделе «Не боритесь с законом Конвея» на с. 228 (глава 9).

Механика

Методы построения эволюционной архитектуры можно использовать в три этапа.

Этап 1. Определите измерения, на которые повлияет эволюция

Прежде всего необходимо определить, какие измерения архитектуры требуется сохранить по мере ее эволюции. Обычно это технические характеристики и такие параметры, как дизайн данных, безопасность, масштабируемость и другие слова на «-ость», которые архитекторы сочли важными. На этом этапе необходимо привлекать другие заинтересованные группы в организации, включая бизнес, группу эксплуатации, группу обеспечения безопасности и др. Полезно использовать *обратный маневр Конвея* (описан в разделе «Не боритесь с законом Конвея» на с. 228, глава 9), поскольку он поощряет создание многофункциональных команд. По сути, это стандартная работа архитектора на старте проекта при определении архитектурных характеристик, которые необходимо поддерживать.

Этап 2. Задайте фитнес-функцию(и) для каждого измерения

Для одного измерения может быть несколько фитнес-функций. Например, для защиты характеристик архитектуры кодовой базы, таких как предотвращение циклов зависимости компонентов, архитекторы обычно подключают набор метрик кода к пайплайну развертывания. Опишите свой выбор того, какие измерения необходимо поддерживать, в легком формате, например в вики. Затем для каждого измерения решите, какие части, эволюционируя, могут демонстрировать нежелательное поведение, и задайте фитнес-функции. Фитнес-функции могут быть автоматическими или ручными и иногда нетривиальными.

Этап 3. Используйте пайплайны развертывания для автоматизации фитнес-функций

Наконец, архитекторы должны обеспечить возможность вносить инкрементные изменения в проект, определив для этого этапы применения фитнес-функций в пайплайне развертывания и управляя процессами развертывания, такими как выделение машин, тестирование и другие задачи DevOps. Инкрементализм — двигатель эволюционной архитектуры, он позволяет эффективно проводить проверку фитнес-функций в пайплайне развертывания и способствует высокой степени автоматизации, чтобы скрыть такие рутинные задачи, как развертывание. Время цикла — это метрика эффективности проектирования в рамках непрерывной доставки. Одна из задач разработчиков в проектах, поддерживающих эволюционную архитектуру, заключается в поддержании оптимального времени цикла. Время цикла — важный показатель инкрементных изменений, поскольку на нем основаны многие другие метрики. Например, скорость появления новых поколений в архитектуре пропорциональна времени цикла. Другими словами, если время цикла проекта увеличивается, это замедляет скорость появления новых поколений, что влияет на способность к эволюции.

Хотя измерения и фитнес-функции определяются в начале работы над проектом, этому необходимо постоянно уделять внимание как в новых, так и в существующих проектах. Для программных продуктов актуальна проблема *неизвестных неизвестных*: разработчики не могут предугадать всё. При проектировании какие-то части архитектуры могут вызывать беспокойство, и фитнес-функции помогают предотвратить распространение этих нарушений. Какие-то фитнес-функции срабатывают уже в начале проекта, но многие не проявят себя, пока в архитектуре не возникнет точка напряжения. Необходимо внимательно следить за ситуациями, в которых нарушаются нефункциональные требования, и добавлять в архитектуру фитнес-функции, чтобы избежать проблем в будущем.

Проекты Greenfield[1]

Встроить эволюционность в новые проекты гораздо проще, чем модернизировать существующие. Во-первых, разработчики могут сразу же вносить инкрементные изменения, создав пайплайн развертывания на начальном этапе проекта. Фитнес-функции лучше определить и спроектировать еще до создания кода, что облегчит их внедрение в сложных случаях, поскольку уже будет иметься поддерживающий код. Во-вторых, архитекторам не приходится распутывать нежелательную связанность, которая появляется в существующих проектах. По мере изменения проекта архитектор также может добавить метрики и другие тесты для обеспечения целостности архитектуры.

Создавать новые проекты, способные к работе при незапланированных изменениях, проще, если выбрать подходящие паттерны и практики эволюционной архитектуры. Например, архитектуру микросервисов отличает чрезвычайно низкая связанность и высокая способность к инкрементным изменениям, что делает этот стиль предпочтительным при построении эволюционных архитектур (и способствует его популярности).

Модернизация существующих архитектур

Добавление способности к эволюции в существующие архитектуры зависит от трех условий: какова степень связанности компонентов, насколько зрелой является команда и легко ли создавать фитнес-функции.

Подходящая связанность (coupling) и связность (cohesion)

Эволюционность технической архитектуры в значительной степени определяется связанностью ее компонентов. Однако даже самая лучшая из эволюционных архитектур обречена, если схема данных в ней жесткая и окаменелая. Чистое разделение систем облегчает эволюцию; избыточная связанность затрудняет. Чтобы строить действительно эволюционные системы, необходимо учитывать все связанные измерения архитектуры.

Помимо технических аспектов связанности, необходимо помнить и о функциональной связности компонентов системы. При миграции архитектуры функциональная связность определяет конечную гранулярность реструктурированных компонентов. Это не значит, что нельзя разделить компоненты на самые мелкие

[1] Greenfield (от англ. «зеленое поле») — проект совершенного нового продукта, разрабатываемого «с нуля» — *Примеч. ред.*

части — они должны иметь размер, соответствующий предметному контексту. Например, связанность в одних бизнес-задачах выше, чем в остальных, как в случае с транзакционными системами. Не стоит пытаться строить максимально разделенную архитектуру, если она не соответствует проблеме.

То, насколько эволюционной будет архитектура, определяют и выбранные инженерные практики. Хотя непрерывная доставка не гарантирует, что архитектура будет эволюционной, без ее применения создать такую архитектуру практически невозможно. Многие команды выбирают усовершенствованные инженерные стратегии, чтобы повысить свою эффективность. Однако освоив эти стратегии, их можно использовать для создания таких продвинутых решений, как эволюционная архитектура. Таким образом, возможность построения эволюционной архитектуры служит стимулом для повышения эффективности.

Многие компании находятся на переходном этапе от старых методов к новым. Они, возможно, уже освоили самые доступные техники, такие как непрерывная интеграция, но до сих пор проводят в основном ручное тестирование. Хотя оно и замедляет время цикла, ручное тестирование важно включать в пайплайны развертывания. Во-первых, все этапы сборки приложения в этом случае принципиально одинаковы — этапы сборки в пайплайне. Во-вторых, по мере распространения автоматизации развертывания ручные этапы автоматизируются естественным образом. В-третьих, разбираясь в каждом этапе, вы лучше узнаете особенности механических процессов сборки и сможете создавать более эффективные циклы обратной связи и другие улучшения.

Самым большим и самым распространенным препятствием для построения эволюционной архитектуры является трудность выполнения операций. Если разработчикам сложно внедрять изменения, это усложнит и весь цикл обратной связи.

РАЗНИЦА МЕЖДУ РЕФАКТОРИНГОМ И РЕСТРУКТУРИЗАЦИЕЙ

Разработчики иногда склонны употреблять звучные термины в более широком смысле, как в случае с *рефакторингом*. По определению Мартина Фаулера, рефакторинг — это реструктуризация существующего компьютерного кода без изменения его внешнего поведения. Многие разработчики употребляют термин «*рефакторинг*» в значении *изменения* вообще, но это не одно и то же.

Рефакторинг архитектуры проводят редко; чаще ее *реструктурируют*, внося существенные изменения в ее структуру и поведение. Паттерны архитектуры существуют в том числе для того, чтобы сделать те или иные ее характеристики основными в приложении. Смена паттернов подразумевает смену приоритетов, и это не рефакторинг. Например, масштабируемость может быть задана методами событийно-управляемой архитектуры. Если команда перейдет на другой паттерн, он, скорее всего, не будет поддерживать тот же уровень масштабируемости.

Мы призываем архитекторов представлять все механизмы проверки в виде фитнес-функций, включая специализированные. Например, во многих архитектурах требования к масштабируемости и соответствующие тесты определяются соглашением об уровне обслуживания, а также существуют специфические требования к безопасности и механизмы их проверки. Архитекторы часто рассматривают эти категории отдельно, но у них одно назначение — проверить какую-то характеристику архитектуры. Если представить все архитектурные проверки как фитнес-функции, то это поможет обеспечить бóльшую согласованность при определении степени автоматизации и других полезных взаимодействий.

Следствия применения технологий COTS

Во многих организациях разработчики не являются владельцами всех элементов экосистемы. В крупных компаниях в основном используются COTS-технологии[1] и инструменты и пакетные программные продукты, что затрудняет создание эволюционных систем.

Системы COTS должны эволюционировать совместно с другими приложениями. К сожалению, способность этих систем к эволюции очень ограниченна. Вот характеристики эволюционной архитектуры, которые плохо поддерживаются в системах COTS:

Инкрементное изменение

Большинство коммерческих программных продуктов не соответствует отраслевым стандартам автоматизации и тестирования. Архитекторы и разработчики часто вынуждены создавать логические барьеры между точками интеграции и встраивать любое возможное тестирование, зачастую рассматривая всю систему как «черный ящик». В таких системах довольно сложно обеспечить гибкость с точки зрения пайплайнов развертывания, DevOps и других современных практик разработки.

Подходящая связанность

Пакетное ПО часто обладает очень плохой связанностью. Как правило, система непрозрачна, с заданным API, который разработчики используют для ее интеграции. Это неизбежно ведет к проблеме, описанной в разделе «Антипаттерн: ловушка последних 10 % и Low Code/No Code» на с. 212 (глава 8), и не обеспечивает достаточной гибкости для выполнения полезной работы.

[1] COTS (англ. commercial off-the-shelf) — готовый к коммерческому использованию. — *Примеч. ред.*

Фитнес-функции

Добавление фитнес-функций в пакетное ПО, пожалуй, самая сложная задача обеспечения эволюционности. Как правило, инструменты такого рода не раскрывают достаточно внутренних компонентов, чтобы можно было провести модульное или компонентное тестирование, поэтому остается только поведенческое интеграционное тестирование. Оно менее предпочтительно, поскольку такие тесты грубые, работают в комплексной среде и тестируют большой участок поведения системы.

Приложите максимум усилий, чтобы удержать точки интеграции на вашем уровне зрелости. Если это невозможно, признайте, что в каких-то частях системы внедрять эволюционные изменения разработчикам будет легче, чем в других.

Еще одна неприятная особенность большого количества пакетных продуктов — непрозрачные экосистемы баз данных. В лучшем случае пакетное ПО полностью управляет состоянием базы данных, раскрывая выбранные значения в точках интеграции. В худшем случае БД поставщика *сама является* точкой интеграции с остальной частью системы, что значительно усложняет изменения по обе стороны API. В этом случае архитекторам и администраторам БД придется «отвоевывать» контроль над базой данных, чтобы получить хоть какую-то надежду на эволюционность.

Имея необходимое пакетное ПО, создавайте набор максимально надежных фитнес-функций и используйте все возможности автоматизировать их выполнение. Отсутствие доступа к внутренним компонентам вынуждает применять менее предпочтительные методы тестирования.

Миграция архитектур

Многие компании со временем меняют выбранные стили архитектуры. Например, в начале развития компании архитекторы выбирают простые архитектурные модели, часто представляющие собой многослойные монолиты. По мере роста компании нагрузка на архитектуру растет. Один из самых частых путей миграции — переход от монолита к архитектуре, основанной на сервисах, обусловленный общим сдвигом в архитектурном мышлении, которое теперь выбирает в качестве основы предметную область, о чем говорится в разделе «Практический пример: микросервисы как эволюционная архитектура» на с. 137 (глава 5). Многие архитекторы прельщаются целью получить в результате миграции высокоэволюционную архитектуру микросервисов, но часто это бывает довольно сложно, в первую очередь из-за существующей связанности.

Когда архитекторы планируют миграцию, они обычно продумывают ее для связанности классов и компонентов, но игнорируют другие измерения, на которые влияет эволюция, например данные. Транзакционная связанность так же реальна, как и связанность между классами, и ее так же легко упустить из виду при реструктуризации архитектуры. Эти внеклассовые точки сцепления становятся тяжкой обузой при попытке разделить существующие модули на мелкие части.

Многие сеньор-разработчики год за годом создают похожие приложения и начинают скучать от однообразия. Большинство предпочитают *написать* фреймворк, а не *использовать* уже готовый: *метаработа интереснее, чем работа*. Работа скучна, рутинна и состоит из повторений, а создавать новое — увлекательно.

Одним из признаков подобной скуки является то, что сеньор-разработчики начинают писать инфраструктуру, которую используют другие разработчики, вместо того чтобы применять существующие инструменты (часто с открытым исходным кодом). У нас был передовой клиент, который создал свой собственный сервер приложений, веб-фреймворк на Java и практически все остальные элементы инфраструктуры. Однажды мы поинтересовались у него, создал ли он собственную операционную систему, и когда он ответил «нет», мы спросили: «Но почему?! Ведь вы же построили с нуля все остальное!»

Можно предположить, что компании требовались возможности, которых не было на рынке. И когда стали доступны инструменты с открытым исходным кодом, у нее уже была своя инфраструктура, созданная с любовью. Компания решила не переходить на стандартный стек, а оставить свой собственный, поскольку различались они незначительно. Через 10 лет все силы лучших разработчиков уходили на его обслуживание, починку сервера приложений, добавление функций в веб-фреймворк и другую рутинную работу. Постоянно занимаясь обслуживанием системы, они не могли использовать инновации для создания лучших приложений.

Архитекторы иногда что-то создают только потому, что это их привлекает или может улучшить их резюме. Конечно, создавать такие *важные* элементы, как фреймворки и библиотеки, интереснее, чем решать мирские проблемы бизнеса, но такова жизнь!

Метаработа интереснее, чем работа.

Не попадайте в ловушку внедрения ради внедрения. Прежде чем встать на этот путь в один конец, убедитесь, что вы рассмотрели и оценили все компромиссы.

Этапы миграции

Многим архитекторам приходится выполнять миграцию устаревшего монолитного приложения на более современную архитектуру, основанную на сервисах. Опытные специалисты понимают, что в приложениях существует множество точек сцепления и первым делом при распутывании кодовой базы необходимо разобраться, как все связано. При разделении монолита необходимо найти подходящий баланс связанности и связности. Например, в архитектуре микросервисов существует строгое ограничение, требующее помещать базу данных внутрь ограниченного контекста сервиса. Даже если разбить классы на достаточно мелкие части реально, разделить транзакционные контексты на такие же части может оказаться невыполнимой задачей.

Многие архитекторы переходят от монолитных приложений к архитектурам на основе сервисов. Рассмотрим исходную архитектуру, представленную на рис. 7.1.

Создавать гранулированные сервисы проще в новых проектах и сложнее при миграции существующих. Итак, как перевести архитектуру на рис. 7.1 в архитектуру на основе сервисов, показанную на рис. 7.2?

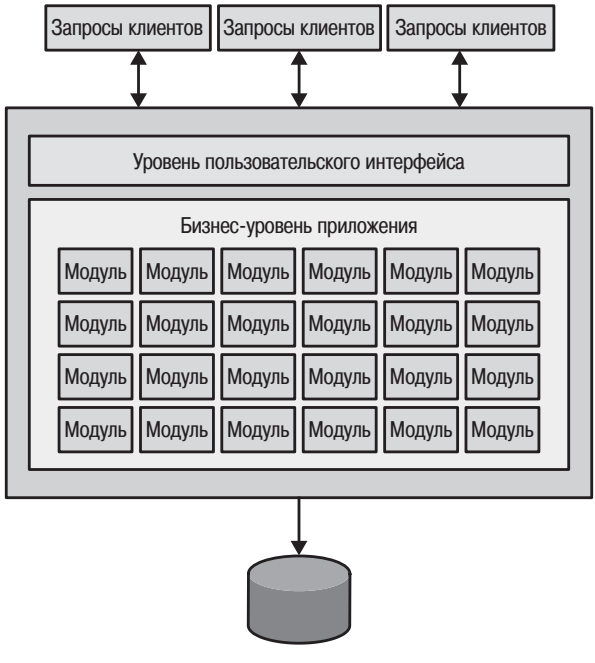


Рис. 7.1. Монолитная архитектура как отправная точка для миграции, архитектура share everything (совместно использовать всё)

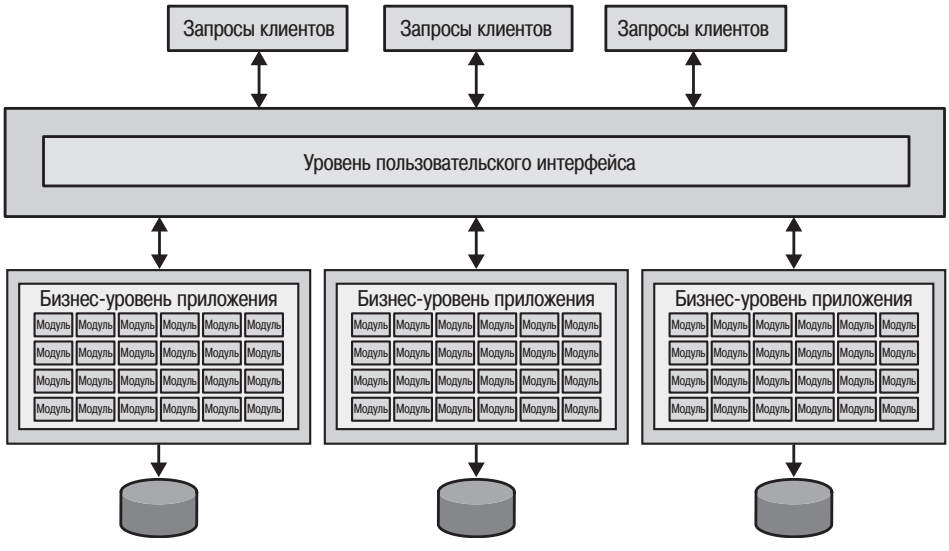


Рис. 7.2. Конечный результат миграции — архитектура на основе сервисов, созданная по принципу «совместно использовать как можно меньше»

Выполнить такую миграцию достаточно сложно. Для этого необходимо учесть целый ряд факторов: гранулярность сервисов, границы транзакций, особенности, связанные с базами данных, а также вопросы организации работы с совместно используемыми библиотеками. Архитекторы должны понимать, почему эта миграция необходима, и причина для нее должна быть более веской, чем просто «потому что все сейчас так делают». Благодаря разбиению архитектуры на предметные области, оптимизации структуры команды и эксплуатационному разделению проводить постепенные изменения становится проще, и это одна из составляющих эволюционной архитектуры, поскольку в этом случае направленность работы соответствует имеющимся рабочим артефактам.

Самое главное при разбиении монолитной архитектуры — обеспечить подходящую гранулярность сервисов. Крупные сервисы облегчают такие задачи, как создание транзакционных контекстов и оркестрация, но не помогают разбить монолит на более мелкие части. Слишком мелкомодульные компоненты ведут к увеличению оркестрации, накладных расходов на взаимодействие и появлению зависимости между компонентами.

На первом этапе миграции архитектуры необходимо определить новые границы сервисов. Разбивать монолит на сервисы можно на основе следующего:

Группы бизнес-функциональности

Задачи бизнеса могут четко соответствовать возможностям ИТ. Создание ПО, имитирующего существующую иерархию бизнес-коммуникаций, точно

соответствует закону Конвея (см. «Не боритесь с законом Конвея» на с. 228 (глава 9).

Границы транзакций

Многие компании должны придерживаться широких границ транзакций. При разбиении монолита архитектуры часто выясняется, что транзакционную связанность труднее всего разделить, как обсуждалось в разделе «Транзакции с двухфазным коммитом» на с. 165 (глава 6).

Цели развертывания

Возможность вносить изменения инкрементно позволяет разработчикам делать релизы по разным графикам. Например, отделу маркетинга обновления могут требоваться гораздо чаще, чем складу. Разделение сервисов по соображениям эксплуатации, таким как скорость выпуска релиза, имеет смысл, если этот критерий очень важен. Точно так же для какой-то части системы можно задать экстремальные эксплуатационные характеристики (например, масштабируемости). Разделение сервисов в соответствии с задачами эксплуатации позволяет разработчикам отслеживать (с помощью фитнес-функций) работоспособность и другие эксплуатационные метрики сервиса.

При более крупной детализации сервисов многие проблемы координации, характерные для микросервисов, исчезают, поскольку внутри одного сервиса находится более широкий бизнес-контекст. Однако чем крупнее сервис, тем больше с ним связано эксплуатационных трудностей (еще один архитектурный компромисс).

Эволюционное взаимодействие модулей

Перенос совместно используемых модулей (включая компоненты) — еще одна частая задача разработчиков. Рассмотрим структуру на рис. 7.3.

На рис. 7.3 все три модуля используют одну библиотеку. Однако необходимо разделить эти модули на отдельные службы. Как сохранить эту зависимость?

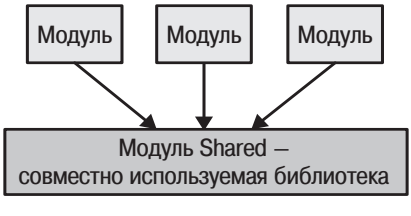


Рис. 7.3. Модули с центробежной и центростремительной связанностью

В некоторых случаях можно разделить библиотеку чисто, сохраняя отдельную функциональность, необходимую каждому модулю. Рассмотрим пример на рис. 7.4.

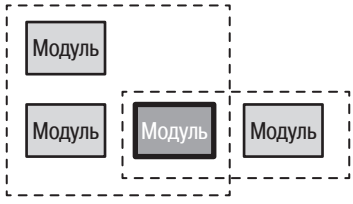


Рис. 7.4. Модули с общей зависимостью

На рис. 7.4 оба модуля нуждаются в конфликтующем модуле, выделенном темной заливкой и жирными границами. Если разработчикам повезет, они разделят функциональность чисто и посередине, разбив общую библиотеку на части, необходимые каждому зависимому модулю, как показано на рис. 7.5.

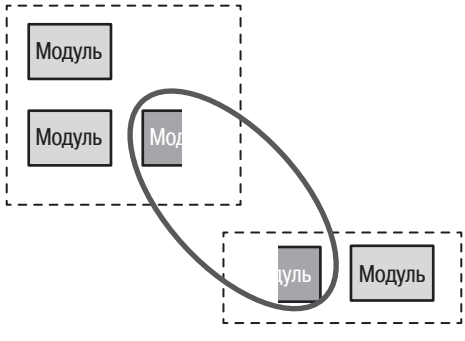


Рис. 7.5. Разделение общей зависимости

У архитекторов не так много полезных метрик на уровне кода, но вот одна из них. Набор метрик Chidamber & Kemerer[1] включает метрики, определяющие, подходит ли модуль для разделения или следует использовать подход под названием «Отсутствие связности в методах» (LCOM, lack of cohesion in methods)[2]. LCOM измеряет структурную связность в классах или компонентах и имеет несколько разновидностей (LCOM1, LCOM2 и т. д.), оценивающих разные параметры. Однако по своей сути эта метрика измеряет *отсутствие* связности. Рассмотрим три случая на рис. 7.6.

[1] https://www.aivosto.com/project/help/pm-oo-ck.html

[2] http://www.cs.sjsu.edu/~pearce/modules/lectures/ood/metrics/lcom.htm

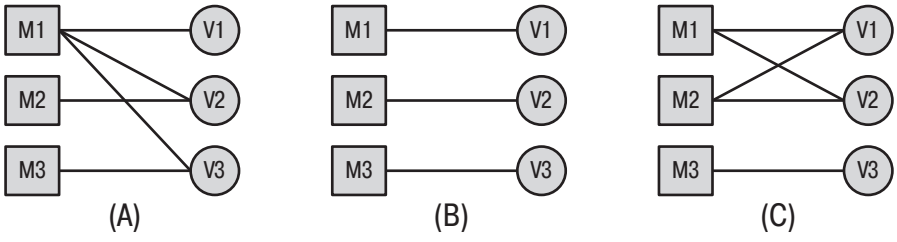


Рис. 7.6. Три класса с разными уровнями связности

На рис. 7.6 *M* обозначает метод, а *V* — поле в классе. В этом примере *A* представляет класс с более высокой связностью (больше методов используют поля), чем *B*, который не имеет связности. На самом деле *B* можно без труда разделить на три отдельных класса.

LCOM измеряет неудачное использование точек сцепления. В примере на рис. 7.6 LCOM в случае *B* выше, чем в случаях *A* или *C*, связность в которых смешанная.

Эта метрика доступна для любой платформы, поддерживающей набор метрик CK; например, распространенная реализация на Java с открытым исходным кодом — ckjm (https://www.spinellis.gr/sw/ckjm/).

LCOM полезен для архитекторов, выполняющих миграцию архитектуры, поскольку основная часть этого процесса связана с совместно используемыми классами или компонентами. При разбиении монолита довольно легко определить, как разделить основные части предметной области. Однако как быть со вспомогательными классами и другими компонентами — насколько они связаны между собой? Например, если при создании монолита в нескольких местах возникает потребность в таком понятии, как `Address`, команда будет использовать один класс `Address`, что вполне логично. Однако что делать с классом `Address`, если этот монолит необходимо разделить? Метрика LCOM помогает определить, должен ли класс вообще быть единым — если значение этой метрики высокое, то класс не обладает связностью. Однако если значение LCOM низкое, архитекторы должны выбрать другой подход.

Остаются два варианта. Первый — извлечь модуль в совместно используемую библиотеку (такую как JAR, DLL, gem и др.) и обращаться к нему из обоих мест, как показано на рис.7.7.

Совместное использование — это форма связанности, которая крайне нежелательна в архитектурах наподобие микросервисов. Альтернативой совместному использованию библиотеки служит репликация, показанная на рис. 7.8.

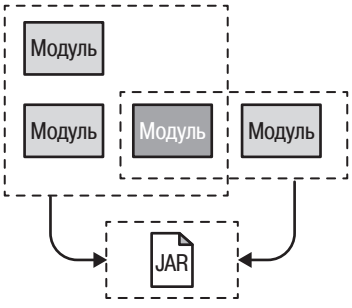


Рис. 7.7. Совместное использование зависимости через JAR-файл

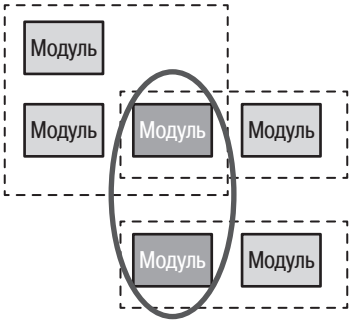


Рис. 7.8. Дублирование совместно используемой библиотеки, позволяющее избавиться от точки сцепления

В распределенной среде разработчики могут реализовать совместное использование с помощью обмена сообщениями или вызова сервисов.

Следующий шаг после разделения сервисов — *отделение* бизнес-слоев от пользовательского интерфейса. Даже в архитектурах микросервисов пользовательские интерфейсы часто сводятся к монолиту — в конце концов, он должен быть единым. Поэтому разработчики обычно разделяют пользовательские интерфейсы на ранних этапах миграции, создавая прокси-слой отображения между компонентами пользовательского интерфейса и внутренними сервисами, к которым они обращаются. При разделении пользовательского интерфейса также создается защитный слой (anticorruption layer), изолирующий изменения пользовательского интерфейса от изменений архитектуры.

Следующий шаг — *обнаружение сервисов*, чтобы сервисы могли находить и вызывать друг друга. Архитектура состоит из сервисов, которые должны быть скоординированы между собой. Создав механизм обнаружения на раннем этапе, разработчики могут постепенно переносить части системы, готовые к изменени-

ям. Обнаружение сервисов часто реализуют как простой прокси-слой: каждый компонент вызывает прокси, который, в свою очередь, обращается к конкретной реализации.

> Все проблемы computer science можно решить, добавив еще один уровень косвенности, кроме, конечно, проблемы избыточной косвенности.
>
> — *Дейв Уилер (Dave Wheeler) и Кевлин Хенни (Kevlin Henney)*

Конечно, чем больше уровней косвенности, тем более сложной становится навигация по сервисам.

При миграции приложения с монолитной архитектуры на архитектуру на основе сервисов необходимо тщательно разобраться, как модули связаны в существующем приложении. Наивное разделение значительно ухудшает производительность. Точки сцепления в приложении становятся точками соединения интеграционной архитектуры, со всеми вытекающими задержками, низкой доступностью и другими проблемами. Более целесообразно разделять монолитную архитектуру на сервисы постепенно, с учетом таких факторов, как границы транзакций, структурная связанность и другие характеристики, проводя реструктуризацию в несколько итераций. Сначала разбейте монолит на несколько крупных частей приложения, скорректируйте точки интеграции — и так несколько раз. В среде микросервисов предпочтительнее проводить миграцию постепенно.

> При переходе от монолита сначала создайте небольшое количество более крупных сервисов.
>
> — *Сэм Ньюман, «Building Microservices»*[1]

Далее необходимо выбрать определенный сервис и отделить его от монолита, фиксируя все точки вызова. Здесь важную роль играют фитнес-функции — их необходимо построить, убедиться, что новые точки интеграции не изменяются, и добавить контракты, управляемые потребителем.

Как создавать эволюционные архитектуры

На протяжении всей книги мы использовали метафоры из области биологии, и вот еще одна. Наш мозг развивался не в комфортной, первозданной среде, где каждая способность формировалась по тщательно выстроенному плану. Новые навыки наслаивались на выработанные ранее. За большую часть наших вегетативных функций (например, дыхание, питание и т. д.) отвечают отделы мозга,

[1] *Ньюман С.* «Создание микросервисов». СПб., издательство «Питер».

несильно отличающиеся от мозга рептилий. Эволюция не заменяет основные механизмы, она создает поверх них новые слои.

Архитектура ПО в крупных организациях эволюционирует по схожей схеме. Большинство компаний предпочитают не создавать функции заново, а адаптировать уже имеющиеся. Как бы мы ни любили рассуждать о чистой, идеальной архитектуре, в реальности она представляет собой хаос из технического долга, конфликтующих приоритетов и ограниченного бюджета. Архитектура в крупных компаниях подобна человеческому мозгу: системы нижнего уровня обрабатывают критические детали, но уже устарели. Компании очень неохотно отказываются от того, что еще работает, а это приводит к серьезным проблемам с интеграцией архитектуры.

Преобразование существующей архитектуры в эволюционную — сложная задача. Если разработчики не встроили в архитектуру свойства, позволяющие легко внедрять изменения, они вряд ли появятся в ней сами собой. Ни один архитектор, каким бы талантливым он ни был, не сможет превратить большой комок грязи в современную архитектуру микросервисов, не приложив огромных усилий. К счастью, необязательно менять всю существующую архитектуру, чтобы заложить в нее способность эволюционировать, — достаточно встроить в нее несколько точек гибкости.

Удаляйте лишнюю изменчивость

Одна из целей непрерывной доставки — обеспечить стабильную разработку из надежных компонентов. Эта цель выражается в современном подходе DevOps к построению неизменяемой инфраструктуры. Мы обсуждали динамическое равновесие экосистемы разработки в главе 1 — и оно особенно заметно в том, насколько сильно смещается фундамент вокруг программных зависимостей. Программные системы постоянно изменяются — разработчики модернизируют возможности, выпускают пакеты обновлений и в целом проводят настройку продукта. Отличный пример — операционные системы, поскольку они постоянно изменяются.

Современный DevOps решил проблему динамического равновесия локально, заменив схемы типа «снежинка» *неизменяемой инфраструктурой*[1] (immutable infrastructure). *Инфраструктура, организованная в виде снежинок* (snowflake infractructure), включает активы, созданные разработчиком вручную, и все ее обслуживание осуществляется тоже вручную. Чад Фаулер (Chad Fowler) ввел термин *«неизменяемая инфраструктура»* в своем блоге, в статье «Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components»

[1] https://www.oreilly.com/radar/an-introduction-to-immutable-infrastructure/

(«Выкиньте все серверы и сожгите код: неизменяемая инфраструктура и одноразовые компоненты»). Неизменяемая инфраструктура относится к системам, определенным полностью программно. Все изменения в системе должны внедряться через код, а не путем модификации работающей ОС. Таким образом, с эксплуатационной точки зрения вся система неизменяема — после ее загрузки изменения не происходят.

Хотя неизменяемость может показаться противоположностью эволюционности, на самом деле все наоборот. Программные системы состоят из тысяч подвижных компонентов, тесно связанных друг с другом. К сожалению, непредвиденных побочных эффектов от изменения одного из них избежать пока не удается. Блокируя возможность непредвиденных изменений, мы берем под контроль больше факторов, влияющих на прочность системы. Чтобы сократить влияние изменений, разработчики стараются заменять переменные в коде константами. С появлением DevOps эта практика стала эксплуатационной и более декларативной.

Неизменяемая инфраструктура соблюдает принцип *удаления ненужных переменных*. Для создания эволюционных программных систем требуется контролировать как можно больше неизвестных. Практически невозможно создать фитнес-функции, способные предсказать, как последний пакет обновлений ОС повлияет на приложение. Вместо этого разработчики создают инфраструктуру заново при каждом выполнении пайплайна развертывания, оперативно выявляя критические изменения. Если разработчики смогут исключить известные основные изменяемые части, такие как операционная система, они смогут облегчить бремя постоянного тестирования.

Архитекторам доступно множество способов преобразования изменяемых объектов в константы. Многие команды распространяют принцип неизменяемости и на среду разработки. Сколько раз вы слышали чей-то возглас: «Но это же работает на моей машине!»? Если у каждого разработчика будет абсолютно одинаковый образ, исчезнет множество ненужных переменных. Например, большинство команд автоматизируют обновление библиотек разработки через репозитории, но как насчет обновления таких инструментов, как IDE? Преобразование среды разработки в неизменяемую позволит каждому члену команды использовать единый фундамент.

Неизменяемая среда разработки также позволяет распространять полезные инструменты на все проекты. Парное программирование — обычная практика в командах agile-разработчиков, оно включает ротацию, когда напарники регулярно, раз в несколько часов или дней, меняются ролями. Однако если на компьютере, который разработчик использовал вчера, был инструмент, которого нет на сегодняшней машине, это затрудняет работу. Создание единого источника обновления позволит легко добавлять полезные инструменты во все системы сразу.

ОПАСНОСТЬ «СНЕЖИНОК»

Случай, описанный в популярном блоге Knightmare: A DevOps Cautionary Tale[1], предупреждает об опасности серверов-«снежинок». В одной финансовой компании существовал алгоритм обработки трейдерских сделок под названием PowerPeg, но уже несколько лет его не использовали. Однако разработчики так и не удалили код. Он скрывался под выключенным переключателем функций. В связи с изменением нормативных требований разработчики внедрили новый алгоритм — SMARS. Они были ленивы и решили для запуска нового кода SMARS использовать старый флаг функции PowerPeg. 1 августа 2012 года новый код был развернут на семи серверах. К сожалению, система работала на восьми серверах — один из них забыли обновить. После включения переключателя функции PowerPeg семь серверов начали продавать, а восьмой — покупать! Разработчики случайно запустили наихудший рыночный сценарий — дешевую продажу и дорогую покупку. Убедившись, что виной всему новый код, разработчики откатили его на семи серверах, но оставили включенным переключатель функций, то есть код PowerPeg теперь работал на всех серверах. На устранение проблемы ушло 45 минут, а убытки составили более 400 миллионов долларов. К счастью, их спас инвестор-ангел, поскольку эта сумма была выше стоимости всей компании.

Эта история подчеркивает проблему неизвестной изменчивости. Повторно использовать старый флаг функции — это безумие; лучшая практика в отношении флагов функции — удалять их, после того как они выполнят свою задачу. В современных средах DevOps безрассудством считается и отсутствие автоматизации развертывания критически важного ПО на серверах.

Делайте решения обратимыми

В активно эволюционирующих системах неизбежно будут возникать непредвиденные сбои. В таких случаях разработчикам приходится создавать новые фитнес-функции, чтобы предотвратить повторные сбои в будущем. Но как восстановить систему после сбоя?

Многие практики DevOps существуют для обеспечения *обратимости решений* — чтобы решение можно было отменить. Среди таких практик, например, *сине-зеленое развертывание* (blue/green deployment), когда развертываются две одинаковые (иногда виртуальные) *экосистемы* — так называемые «*синяя*» и «*зеленая*». Если рабочая система *синяя*, то *зеленая* является подготовительной для следующего релиза. Когда *зеленый* релиз готов, он становится рабочей системой, а *синяя* временно переходит в статус резервной. Если что-то пойдет не так с *зеленой* системой, можно без особых проблем вернуться к *синей*. Если с *зеленой* все в порядке, *синяя* становится подготовительной для следующего релиза.

[1] https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

Переключатели функций (feature toggles) — еще один распространенный способ сделать решения обратимыми. Развертывая изменения под переключателями функций, разработчики могут выпустить их для небольшого пула пользователей (так называемый канареечный релиз), чтобы проверить их работу. Если функция поведет себя неожиданно, разработчики могут переключиться обратно и исправить ошибку, а затем повторить попытку. Не забудьте удалить устаревшие переключатели!

Использование переключателей функций значительно снижает риск в канареечных сценариях. Сервисная маршрутизация — направление к определенному экземпляру сервиса на основе контекста запроса — еще один популярный метод развертывания канареечных релизов в экосистемах микросервисов.

Выбирайте эволюционность, а не предсказуемость

> ...потому что, как мы знаем, существуют известные известные. Это вещи, о которых мы знаем, что мы их знаем. Существуют известные неизвестные. Это вещи, о которых мы знаем, что мы их не знаем. Но еще существуют неизвестные неизвестные. Это вещи, о которых мы не знаем, что мы их не знаем.
>
> — *Дональд Рамсфелд (Donald Rumsfeld), бывший министр обороны США*

Неизвестные неизвестные — заклятый враг программных систем. Многие проекты начинаются с составления списка *известных неизвестных*: того, что разработчики знают, что они должны узнать о сфере и технологии. Однако проекты также становятся жертвами *неизвестных неизвестных*: вещей, о которых никто не знал, что они появятся, но которые неожиданно появились. Именно поэтому все усилия Big Design Up Front[1] оказываются напрасны — архитекторы не могут учесть неизвестные.

> Все архитектуры становятся итерационными из-за *неизвестных неизвестных*; agile просто признает это и делает это раньше.
>
> — *Марк Ричардс*

Хотя ни одна архитектура не способна существовать в условиях неизвестности, мы знаем, что динамическое равновесие нивелирует пользу предсказуемости. Гораздо эффективнее создавать *эволюционные* продукты: если изменения вносить легко, хрустальный шар не нужен. Архитектуру целиком невозможно создать

[1] Big Design Up Front (от англ. «большой проект заранее») — инженерный подход, при котором разработка продукта полностью предшествует реализации. Это делает его похожим на водопадную модель. — *Примеч. ред.*

заранее — проекты постоянно меняются, и эти изменения могут быть ожидаемыми, а могут быть и нет. Чтобы защититься от влияния изменений, разработчики часто проектируют *защитный слой* (anticorruption layer).

Создавайте защитный слой

Время от времени приходится подключать проекты к библиотекам, предоставляющим дополнительные возможности: очереди сообщений, системы поиска и др. *Антипаттерн Abstraction Distraction* (Отвлечение абстракции) описывает сценарий, когда проект слишком сильно «привязывает» себя к внешней библиотеке, коммерческой или с открытым исходным кодом. Когда приходит время обновлять или менять библиотеку, в большую часть кода приложения, использующего библиотеку, оказываются встроены предположения, основанные на библиотечных абстракциях. Предметно-ориентированное проектирование предусматривает защиту от этого явления, называемую *защитным слоем*. Вот пример.

Чтобы избежать слишком раннего усложнения системы, agile-архитекторы при принятии решений руководствуются принципом *последнего ответственного момента*. Мы работали над проектом на Ruby on Rails для клиента, который занимался оптовой продажей автомобилей. После запуска приложения возник неожиданный рабочий процесс. Оказалось, что продавцы подержанных автомобилей добавляли информацию о машинах на сайт аукциона большими партиями как по количеству автомобилей, так и по количеству фотографий на каждый автомобиль. Мы поняли, что не только покупатели не доверяют дилерам подержанных авто — дилеры не доверяют друг другу, поэтому фотографируют мельчайшие детали. Пользователям требовался механизм, показывающий состояние загрузки файлов — например, индикатор прогресса — либо позволяющий позже узнать, загружена ли вся партия. В переводе на технический язык им нужна была асинхронная загрузка.

Обычно эта проблема решается с использованием очереди сообщений, и команда обсуждала, стоит ли добавить в архитектуру очередь с открытым исходным кодом. На этом этапе многие проекты часто попадают в ловушку: «Мы знаем, что нам обязательно будет нужна очередь сообщений, поэтому давайте купим самую красивую, а потом ее доработаем». Проблема тут в *техническом долге*: в том, что есть в проекте, хотя его там быть не должно, и что мешает тому, что должно там быть. Большинство разработчиков уверены, что технический долг — это только старый некачественный код, но технический долг может образоваться из-за преждевременного усложнения проекта.

Архитектор попросил разработчиков поискать более простой способ. Один из них обнаружил BackgrounDRb (https://github.com/gnufied/backgroundrb), очень простую библиотеку с открытым исходным кодом, которая имитирует очередь сообщений,

поддерживаемую реляционной базой данных. Архитектор знал, что этот простой инструмент, вероятно, никогда не будет масштабироваться для решения других задач, но других возражений у него не было. Таким образом, вместо попыток предсказать будущее использование, его сделали относительно легко заменяемым, поместив за API. В *последний ответственный момент* ответьте на следующие вопросы: нужно ли принимать это решение сейчас?, можно ли безопасно отложить этот вопрос, не замедляя работу? и что можно установить сейчас в качестве временного решения, что можно будет легко изменить при необходимости?

Примерно через год вновь потребовалось реализовать асинхронность для событий продаж с привязкой ко времени. Архитектор оценил ситуацию и решил, что второго экземпляра BackgrounDRb будет достаточно. Еще через год поступил запрос на постоянное обновление значений, таких как кэш и сводки. Команда поняла, что существующее решение не справится с новой нагрузкой. Однако она уже четко представляла, какое асинхронное поведение необходимо приложению. В итоге проект перешел на Starling (https://github.com/starling/starling), простую, но традиционную очередь сообщений. Поскольку исходное решение было изолировано за интерфейсом, пара разработчиков завершила переход менее чем за одну итерацию (в этом проекте длившуюся неделю), не нарушая работу других.

Поскольку архитектор предусмотрел защитный слой, замена части функциональности стала чисто механической задачей. При создании защитного слоя на первый план выходит *семантика* конструкций библиотеки, а не *синтаксис* конкретного API. Но это не повод *абстрагировать все*! Некоторые разработчики обожают упреждающие слои абстракции, но непонятно, почему вы должны вызвать `Factory`, чтобы получить `proxy` для удаленного интерфейса к `Thing`. К счастью, большинство современных языков и IDE позволяют извлекать интерфейсы *точно вовремя*. Если проект связан с устаревшей библиотекой, которую нужно изменить, IDE может *извлечь интерфейс* от имени разработчика, создав защитный слой JIT (Just In Time — «точно вовремя»).

Чтобы оградить проект от изменений в библиотеке, создавайте защитные слои JIT.

Одна из первоочередных обязанностей архитектора — контролировать точки сцепления в приложении, особенно с внешними ресурсами. Старайтесь добавлять зависимости рационально. Как архитектор помните, что зависимости не только предоставляют преимущества, но и накладывают ограничения. Убедитесь, что преимущества перевешивают затраты на обновления, управление зависимостями и т. д.

> Разработчики замечают преимущества, но не видят компромиссов!
>
> — *Рич Хикки (Rich Hickey), создатель Clojure*

Архитекторы должны оценивать как преимущества, так и компромиссы и подбирать соответствующие практики разработки.

Защитные слои помогают создавать эволюционные системы. Хотя мы не способны предсказывать будущее, с помощью защитного слоя мы можем по крайней мере снизить стоимость изменений, чтобы их влияние не оказалось разрушительным.

Создавайте жертвенные архитектуры

В книге «Мифический человеко-месяц» Фред Брукс отмечает, что при создании новой программной системы нужно «планировать выбросить одну».

> Проблема не в том, создавать или нет опытную систему, которую придется выбросить. Вы все равно это сделаете. [...] Поэтому планируйте выбросить первую версию — вам все равно придется это сделать.
>
> — *Фред Брукс*

Его мысль заключалась в том, что после создания системы становятся понятны все неизвестные неизвестные, а также правильные архитектурные решения, которые никогда не бывают ясными с самого начала, — а в следующей версии можно реализовать все выявленные преимущества. Разработчикам трудно предвидеть радикально меняющиеся требования и характеристики архитектуры. Один из способов собрать данные для выбора правильной архитектуры — построить ее пробную версию. Мартин Фаулер определяет *жертвенную архитектуру* (sacrificial architecture)[1] как архитектуру, которую удаляют, если модель окажется успешной. Например, eBay начинался с набора скриптов на Perl в 1995 году, затем перешел на C++ в 1997 году и на Java в 2002 году. Очевидно, что такая неоднократная перестройка не помешала eBay стать чрезвычайно успешным. Twitter — еще один хороший пример удачного использования этого подхода. Изначально он был написан на Ruby on Rails, чтобы как можно быстрее выйти на рынок. Однако по мере роста популярности платформа перестала справляться с нагрузкой, что привело к частым сбоям и ограниченной доступности. Для первых пользователей видеть символ сообщения об ошибке стало привычным делом — см. рис. 7.9.

[1] https://martinfowler.com/bliki/SacrificialArchitecture.html

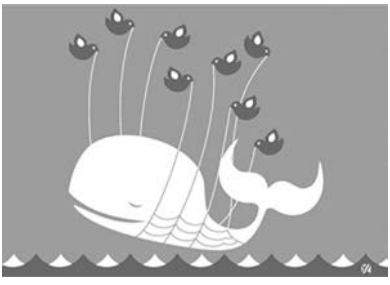


Рис. 7.9. Знаменитый кит, сопровождающий сообщение об ошибке в Twitter

Twitter реструктурировал свою архитектуру, заменив бэкенд более надежным инструментом. Можно утверждать, что именно это помогло компании выжить. Если бы инженеры Twitter с самого начала создали надежную платформу, это задержало бы их выход на рынок и их бы опередил *Snitter* или иной альтернативный сервис обмена короткими сообщениями. Несмотря на сложности роста, жертвенная архитектура в конечном итоге оправдала себя.

Облачные среды добавляют жертвенной архитектуре преимуществ. Если у разработчиков есть проект, который нужно протестировать, развертывание начальной версии в облаке поможет значительно сократить требуемые ресурсы. Если проект окажется успешным, архитекторы смогут потратить образовавшийся ресурс времени на оптимизацию архитектуры. Грамотное внедрение защитных слоев и других практик эволюционной архитектуры позволит смягчить неудобства миграции.

Часто жертвенную архитектуру создают для разработки минимального жизнеспособного продукта (MVP, minimum viable product), чтобы проверить, существует ли на него спрос. Хотя это хорошая стратегия, в конечном итоге придется выделить время и ресурсы на создание более надежной архитектуры — желательно не так заметно, как Twitter.

Еще одна разновидность технического долга возникает во многих изначально успешных проектах. Фред Брукс говорит о нем как о *синдроме второй системы* — когда небольшие, элегантные и успешные системы вследствие завышенных ожиданий превращаются в гигантские чудовища, перегруженные функциональностью. Бизнес не любит удалять то, что работает, поэтому компоненты архитектуры только добавляются, но никогда не удаляются и не выводятся из эксплуатации.

Технический долг служит также отличной метафорой общих недостатков в дизайне проекта независимо от причин, по которым они возникли. Технический долг усугубляет проблему излишней связанности: плохой дизайн часто

проявляется в виде сильной связанности и других антипаттернов, которые затрудняют реструктуризацию кода. Проводя ее, первым делом избавьтесь от унаследованных компромиссов в дизайне, которые образуют технический долг.

Смягчайте влияние внешних изменений

Любая платформа имеет *внешние зависимости*: инструменты, фреймворки, библиотеки и другие активы, предоставляемые и (что более важно) обновляемые через интернет. Программные продукты образуются из растущей стопки абстракций, каждая из которых основывается на предшествующих. Например, операционные системы — это внешняя зависимость, которую разработчик не контролирует. Если компании не хотят писать собственную ОС и остальной вспомогательный код, им придется полагаться на внешние зависимости.

Большинство проектов зависят от множества сторонних компонентов, внедряемых с помощью инструментов сборки. Разработчикам нравятся зависимости, потому что они полезны, но многие не осознают, что за эту пользу приходится платить. Используя сторонний код, необходимо создавать собственные средства защиты от непредвиденных ситуаций: критических изменений, необъявленного удаления и др. Управление внешними зависимостями имеет решающее значение для создания эволюционной архитектуры.

11 СТРОК КОДА, КОТОРЫЕ СЛОМАЛИ ИНТЕРНЕТ

В начале 2016 года разработчикам JavaScript был преподан урок того, как опасно зависеть от мелочей. Один разработчик, создавший много мелких утилит и недовольный тем, что владельцы коммерческого продукта попросили его переименовать один из его модулей, удалил более 250 пакетов, в том числе библиотеку под названием `left pad` , которая содержала 11 строк кода для заполнения строки нулями или пробелами (если 11 строк кода можно назвать библиотекой). К сожалению, многие крупные проекты JavaScript (включая `node.js`) использовали эту зависимость. Когда она исчезла, все развертывания JavaScript сломались.

Администратор репозитория пакетов JavaScript предпринял беспрецедентный шаг — восстановил удаленный код, но ситуация вызвала в сообществе множество споров о том, насколько разумно вообще использовать зависимости.

Из этой истории можно сделать два ценных вывода. Во-первых, помните, что внешние библиотеки *не только* обеспечивают преимущества, но и влекут за собой накладные расходы. Убедитесь, что преимущества оправдывают расходы. Во-вторых, ограждайте сборки от влияния внешних сил. Если необходимая зависимость внезапно исчезнет, откажитесь от нее.

В своем письме от марта 1968 года «О вреде оператора GO TO» («Go To Statement Considered Harmful») редактору журнала *Communications of the ACM* Эдсгер Дейкстра, легенда информатики, раскритиковал принятую тогда лучшую практику написания неструктурированного кода, что в конечном итоге привело к революции структурного программирования. С тех пор фраза considered harmful («признано вредным») стала мемом разработки.

> Управление переходными зависимостями — это наш «considered harmful».
>
> — *Крис Форд (не родственник Нила)*

Крис говорит о том, что пока мы не признáем серьезность проблемы, мы не сможем найти ее решение. Хотя мы и не нашли решения, нам необходимо привлекать внимание к этой проблеме, поскольку она критически влияет на эволюционную архитектуру. Стабильность — одна из основ как непрерывной доставки, так и эволюционной архитектуры. Невозможно строить повторно используемые инженерные практики на основе неопределенности. Разрешение третьим лицам вносить изменения в основные зависимости противоречит принципу стабильности.

Мы рекомендуем разработчикам управлять зависимостями проактивно. Начать стоит с моделирования внешних зависимостей с помощью *вытягивания* (pull model). Например, создайте внутренний репозиторий контроля версий, который будет служить хранилищем компонентов сторонних разработчиков, и рассматривайте изменения из внешнего мира как запросы на внесение изменений (пул-реквестов) в этот репозиторий. Если изменение полезное, внедрите его в экосистему. Однако если основная зависимость внезапно исчезнет, отклоните запрос.

В рамках концепции непрерывной доставки репозиторий сторонних компонентов использует собственный пайплайн развертывания. Когда происходит обновление, пайплайн развертывания внедряет изменение, затем выполняет сборку и проводит smoke-тест затронутых приложений. В случае успеха изменение допускается в экосистему. Таким образом, сторонние зависимости используют те же практики и механизмы разработки, что и внутренние, нивелируя зачастую неважные различия между собственным кодом и сторонними зависимостями — в конце концов, это код одного проекта.

Обновление библиотек и фреймворков

В быту архитекторы часто разграничивают *библиотеки* и *фреймворки*, говоря, что «код разработчика вызывает библиотеку, а фреймворк вызывает код разработчика». Как правило, разработчики создают подклассы из фреймворков (вызыва-

ющих эти производные классы), поэтому фреймворк вызывает код. И наоборот, код библиотеки обычно представляет собой набор связанных классов и/или функций, которые разработчики вызывают по мере необходимости. Поскольку фреймворк вызывает код, этот код имеет сильную связанность с фреймворком в отличие от кода библиотеки, который обычно более утилитарный (например, парсеры XML, сетевые библиотеки и т. д.) и имеет более слабую связанность.

Библиотеки предпочтительнее, потому что они добавляют в приложение меньше связанности, и поэтому их легче заменить при необходимости.

Старайтесь использовать библиотеки, а не фреймворки.

Одна из причин различия в работе с библиотеками и фреймворками заключается в разных применяемых с ними практиках разработки. Фреймворки включают такие возможности, как пользовательский интерфейс, объектно-реляционное отображение, строительные леса типа «модель — представление — контроллер» и т. д. Поскольку фреймворк — основа всего приложения, изменение фреймворка затрагивает весь код. Многие знают: если основной фреймворк устарел более чем на две версии, то для его обновления приходится прикладывать титанические усилия.

Поскольку фреймворк — фундаментальная часть приложения, его регулярное обновление строго обязательно. Библиотеки обычно образуют менее хрупкие точки сцепления, чем фреймворки, что позволяет подходить к их обновлениям с меньшей строгостью. Одна из неформальных моделей управления рассматривает обновления фреймворков как push-обновления, а обновления библиотек — как pull-обновления. Обновлять фундаментальный фреймворк (такой, в котором центробежная/центростремительная связанность выше определенного порога) необходимо, как только новая версия станет стабильной и будут подходящие условия для внедрения изменений. Чем раньше провести обновление, тем меньше времени и усилий оно займет.

Поскольку большинство библиотек предоставляют утилитарную функциональность, их можно обновлять, только когда в них появляется новая требуемая функция, используя модель «обновлять по мере необходимости» (update when needed).

Зависимости фреймворка необходимо обновлять агрессивно; для библиотек допускается пассивное обновление.

Выбирайте внутреннее версионирование

По мере развития любой интеграционной архитектуры неизбежно приходится версионировать конечные точки сервисов. Существует два распространенных шаблона: *нумерация версий*, или *внутреннее разрешение* (internal resolution). В первом случае при критическом изменении разработчики создают новое имя конечной точки, часто включающее номер версии. Таким образом, существующие точки интеграции могут вызывать старую версию, а новые — новую. Второй вариант — внутреннее разрешение, когда вызывающие стороны не меняют конечную точку; в нее встраивают логику для определения контекста вызывающей стороны, чтобы конечная точка возвращала правильную версию. Преимущество сохранения имени в том, что вызывающие приложения не привязаны к конкретным номерам версий.

В любом случае строго ограничивайте количество поддерживаемых версий. Чем больше версий, тем больше тестов и другой нагрузки. Старайтесь поддерживать одновременно только две версии и только на непродолжительный срок.

При версионировании сервисов старайтесь использовать внутреннее версионирование, а не нумерацию; поддерживайте только две версии одновременно.

Практический пример: эволюция рейтингов PenultimateWidgets

PenultimateWidgets имеет архитектуру микросервисов, поэтому разработчики могут изменять ее небольшими частями. Рассмотрим подробнее одно из таких изменений — переход на новый сервис звездных рейтингов, описанный в главе 3. Текущий сервис звездного рейтинга PenultimateWidgets выглядит так: см. рис. 7.10.

Как показано на рис. 7.10, сервис звездного рейтинга включает базу данных и архитектуру с уровнями персистентности, бизнес-правил и пользовательского интерфейса. Не все микросервисы PenultimateWidgets содержат пользовательский интерфейс. Некоторые сервисы в первую очередь информационные, а пользовательский интерфейс других тесно связан с поведением сервиса, как в случае со звездным рейтингом. Данные содержатся в традиционной реляционной БД, которая включает столбец для отслеживания рейтингов по ID элемента.

Когда было решено обновить сервис для поддержки рейтинга в половину звезды, исходная конфигурация была изменена, как показано на рис. 7.11.

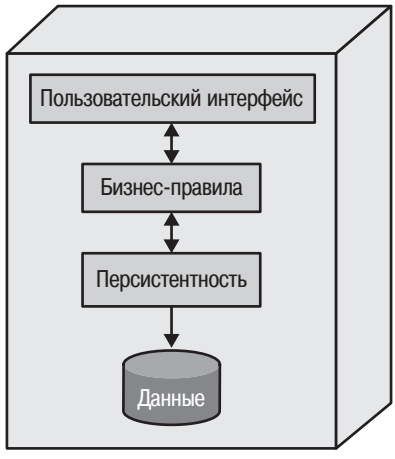


Рис. 7.10. Внутренняя схема сервиса звездного рейтинга PenultimateWidgets

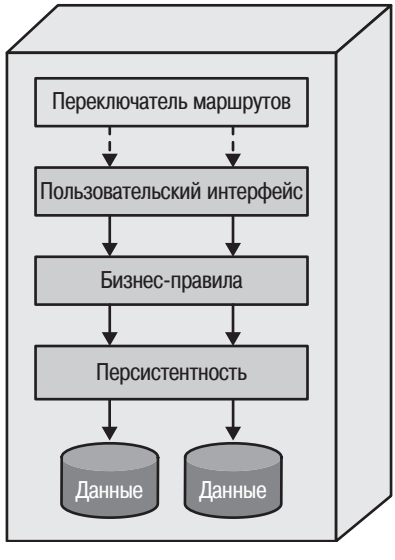


Рис. 7.11. Этап перехода, на котором сервис StarRating поддерживает оба типа рейтингов

На рис. 7.11 показано добавление в базу данных нового столбца с данными о том, есть ли у рейтинга дополнительная полузвезда. Архитекторы также добавили прокси для определения различий в запросах на границе сервиса. Сервис звездного рейтинга не заставляет вызывающие службы «понимать» номер своей версии, а определяет тип запроса, отправляя ответ в любом случае. Это пример использования *маршрутизации* как эволюционного механизма. Сервис

звездного рейтинга может работать так все время, пока какие-то сервисы еще используют звездные рейтинги.

После того как все зависимые сервисы перейдут на новый тип рейтинга с половиной звезды, разработчики могут удалить старый маршрут, как показано на рис. 7.12.

Кроме старого маршрута можно удалить прокси-слой, определяющий различия версий (или оставить его для будущих изменений).

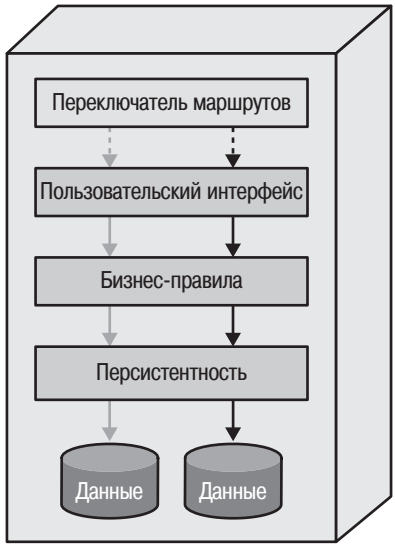


Рис. 7.12. Конечное состояние сервиса StarRating, поддерживающее только новый тип рейтинга

В данном случае с точки зрения эволюции данных изменение не было сложным, поскольку разработчики смогли внести его аддитивно, то есть дополнить схему базы данных, а не изменить ее. Но как быть в случае, когда, чтобы добавить новую функцию, необходимо изменить и базу данных? Ответ на этот вопрос дан в главе 6 при обсуждении эволюционного проектирования структур данных.

Архитектура на основе фитнес-функций

Одной из распространенных практик agile-методологии является *разработка на основе тестов* (test-driven development), когда написанию функциональности предшествует написание модульных тестов. Аналогичный процесс можно использовать в архитектуре, особенно когда успешная работа приложения строго

зависит от определенных возможностей. Создание фитнес-функции, управляющей этой возможностью, гарантирует, что проектирование других частей архитектуры будет основываться на сохранении именно этой возможности.

Известен пример создателей архитектуры LMAX (https://martinfowler.com/articles/lmax.html), которые использовали этот подход. Изменение законодательства, регулирующего рынки в конкретной стране, позволило обычным гражданам совершать сделки онлайн (покупать и продавать) без специальной лицензии. Однако для успешной работы приложение должно было обрабатывать миллионы транзакций в секунду. По разным причинам в качестве технологической платформы была выбрана Java, которая по умолчанию не отличалась масштабируемостью на таком уровне. Таким образом, первым делом архитекторы задали фитнес-функцию, измеряющую скорость транзакций, и начали пробовать различные варианты дизайна для достижения нужного значения. Они начали с потоков, но и близко не смогли подобраться к нужным цифрам. Затем они попробовали различные реализации модели акторов, но тоже не приблизились к цели. Измерив все части системы, они поняли, что бизнес-логика составляла крошечный процент времени вычислений — все остальное занимало переключение контекста.

Вооружившись этими знаниями, они разработали подход, известный как входные и выходные деструкторы (input/output disruptors)[1], который использовал один поток и круговые буферы, чтобы достичь скорости обработки более шести миллионов транзакций в секунду на одном потоке. Архитектура подробно описана на сайте https://martinfowler.com/articles/lmax.html (и многие ее части являются открытыми).

В ходе работы команда использовала термин *mechanical sympathy* («чувство машины») по отношению к аппаратному и программному обеспечению — один из архитекторов был поклонником гонок Формулы-1. Действительно, великие пилоты Формулы-1 «чувствуют» свою машину — они знают, как работает каждая деталь, и интуитивно понимают, все ли в порядке. В программировании «чувствовать машину» — значит понимать, какие слои лежат под абстракциями и на чем основана каждая характеристика, например производительность. В случае с обработкой последовательности запросов/ответов на что уходит больше всего времени на каждом уровне, вплоть до сетевого, и как это можно оптимизировать?

Mechanical sympathy требует введения фитнес-функций как для определения целей, так и для управления строгими требованиями по мере их изменений. Когда команда LMAX достигла своей первоначальной цели, она оставила фитнес-функции работать, меняя только направления, если фитнес-функции противоречили используемому подходу.

[1] https://martinfowler.com/articles/lmax.html#InputAndOutputDisruptors

Многие команды разработчиков начали применять такой подход, как архитектура на основе фитнес-функций, особенно в ситуациях, подобных описанной выше, когда успех зависит от достижения целевого показателя характеристики архитектуры. Так же как и в разработке на основе тестов, в архитектуре на основе фитнес-функций изменения не влияют на критерии успеха.

Итоги

Составляющие эволюционной архитектуры нельзя рассматривать по отдельности — чтобы построить эволюционирующую архитектуру, необходимо обеспечить взаимодействие фитнес-функций и структуры.

Для того чтобы такие практики, как непрерывная интеграция и разработка на основе тестов, стали применяться повсеместно, потребовалось много лет. Зачастую архитекторы применяют инструменты эволюционной архитектуры — мониторы, специальные метрики и периодические проверки, но продолжают использовать устаревшие методы управления, такие как советы по архитектуре, код-ревью и другие доказанно неэффективные практики.

Если архитекторы хотят создавать системы, способные пережить множество изменений как в предметной области, так и в технологическом стеке, они должны создавать фитнес-функции и управлять связанностью с помощью контрактов, чтобы получать полезную обратную связь о важных показателях. Когда в проекте меняются какие-то из нескольких тысяч его составляющих, архитекторы должны быть уверены, что вся система по-прежнему работает правильно. И эту уверенность обеспечивает эволюционная архитектура.

ГЛАВА 8

Подводные камни и антипаттерны эволюционной архитектуры

Мы не раз возвращались к важности выбора подходящего уровня связанности в архитектурах. Однако мы живем в реальном мире и встречаем связанность, которая *вредит* эволюционности проектов.

В программных проектах встречается два вида плохих практик — *подводные камни* и *антипаттерны*. Многие разработчики используют слово «*антипаттерн*» как жаргонный синоним плохого, но его значение несколько тоньше. Во-первых, антипаттерн — это практика, которая вначале выглядела подходящей, но оказалась ошибочной. Во-вторых, для большинства антипаттернов существуют лучшие альтернативы. Многие антипаттерны становятся очевидны только задним числом, поэтому их трудно избежать. *Подводные камни* выглядят как хорошая идея, но то, что это плохой путь, становится понятно сразу же. В этой главе мы рассмотрим и подводные камни, и антипаттерны.

Техническая архитектура

В этом разделе мы разберем распространенные в отрасли практики, которые вредят способности команд развивать архитектуру.

Антипаттерн: ловушка последних 10 % и Low Code/No Code

В свое время Нил был техническим директором консалтинговой фирмы, которая создавала проекты для клиентов, используя инструменты четвертого поколения (4GL), в том числе Microsoft Access. Он добился прекращения использования Access и всех таких инструментов после того, как заметил, что каждый проект Access начинался успешно, но заканчивался неудачей, и захотел понять почему.

Они с коллегой выяснили, что в Access и других популярных в то время инструментах 4GL 80 % потребностей клиента удовлетворялись легко и быстро. Эти среды моделировались как инструменты быстрой разработки приложений, с поддержкой перетаскивания (drag and drop) в пользовательском интерфейсе и другими полезными штуками. Однако следующие 10 % потребностей клиента удовлетворить было хоть и можно, но крайне сложно, потому что требуемая функциональность не была встроена в инструмент, фреймворк или язык. Поэтому умные разработчики придумали способ взломать инструменты: добавлением скрипта для выполнения там, где ожидалось статическое действие, образованием цепочек методов и так далее. Это дало возможность удовлетворять от 80 до 90 % потребностей. Инструмент все равно не мог решить проблему полностью — и это стало называться *ловушкой последних 10 %* (Last 10 % Trap), из-за которой проекты терпели неудачу. 4GL-инструменты позволяли быстро создавать простые вещи, но они не масштабировались, чтобы соответствовать требованиям реального мира. Разработчики вернулись к языкам общего назначения.

Ловушка последних 10 % периодически проявляется в инструментах, призванных устранить сложность разработки и (якобы) обеспечить полнофункциональную разработку с предсказуемыми результатами. В современной разработке она актуальна для сред *low-code/no-code*[1], от full-stack-разработки до специализированных инструментов, таких как оркестраторы.

Хотя в средах low-code нет ничего плохого, их значение почти повсеместно преувеличивают, считая панацеей для разработки благодаря высокой скорости доставки. Их можно применять для решения специализированных задач, но необходимо осознавать существующие ограничения и уметь определять, какое влияние эти ограничения окажут на экосистему.

Обычно, экспериментируя с новым инструментом или фреймворком, разработчики создают простой проект Hello, World. В средах low-code простые вещи чрезвычайно просты, и, наоборот, необходимо знать, что в них сделать *невозможно*. Таким образом, на этапе тестирования инструмента вместо создания простых проектов постарайтесь найти пределы его функциональности, чтобы предусмотреть альтернативы для случаев, с которыми он не справится.

Работая с инструментами low-code/no-code, в первую очередь оценивайте *самые трудные* задачи, а не самые простые.

[1] Low-code/no-code — методы разработки, позволяющие создавать приложения из готовых блоков, с минимальным написанием кода вручную (low-code) или вообще без написания кода (no-code). — *Примеч. ред.*

Практический пример: повторное использование в PenultimateWidgets

Административная функциональность PenultimateWidgets предусматривает ввод данных в специализированную сетку. Поскольку это представление требуется в нескольких местах, PenultimateWidgets решила создать многократно используемый компонент, включающий пользовательский интерфейс, валидацию и другие полезные модели поведения по умолчанию. Используя этот компонент, можно легко создавать новые, подробные интерфейсы администрирования.

Однако ни одно архитектурное решение не обходится без компромиссов. Со временем команда разработчиков компонента превратилась в силос внутри организации, поглотивший нескольких лучших программистов PenultimateWidgets. Команды, использующие компонент, должны запрашивать новые функции у команды разработчиков компонента, которая оказалась завалена исправлениями ошибок и запросами функций. Хуже того, базовый код не соответствует современным веб-стандартам, что затрудняет или делает невозможным внедрение новых функций.

Хотя архитекторы PenultimateWidgets реализовали повторное использование, в конечном итоге это привело к возникновению узкого места. Одно из преимуществ повторного использования в том, что оно позволяет быстро создавать новые функции. Однако если разработчики компонентов не будут успевать за темпами инноваций в условиях динамического равновесия, повторное использование компонентов технической архитектуры обречено стать антипаттерном.

Мы не предлагаем отказаться от создания многократно используемых активов — скорее необходимо постоянно проверять, что они по-прежнему полезны. В случае с PenultimateWidgets, как только архитекторы поняли, что компонент является узким местом, они разорвали точку сцепления. Они разрешили всем, кому это требовалось, самостоятельно добавлять новые функции в код компонента (при условии, что команда разработчиков приложения поддерживала предлагаемые изменения), и перевели все команды, желавшие отказаться от прежнего подхода, на новую модель.

Из опыта PenultimateWidgets можно сделать два вывода. Во-первых, если точки сцепления препятствуют эволюции или ухудшают важные характеристики архитектуры, необходимо разорвать связанность путем ветвления или дублирования.

В PenultimateWidgets связанность разорвали, позволив командам самим отвечать за общий код. Хотя это и увеличило их нагрузку, но одновременно и повысило производительность. В других случаях, как вариант, можно абстрагировать часть общего кода от более крупного фрагмента, чтобы добиться выборочной связанности и частичного разделения.

Во-вторых, необходимо постоянно оценивать пригодность характеристик архитектуры, чтобы убедиться, что они по-прежнему эффективны и не превратились в антипаттерны.

Слишком часто случается так, что решение, которое было правильным на момент своего принятия, со временем становится не лучшим выбором из-за изменившихся условий, таких как динамическое равновесие. Например, архитекторы разрабатывают систему как настольное приложение, но развитие отрасли вынуждает их создавать веб-версию, поскольку привычки пользователей меняются. Исходное решение не было ошибочным, но экосистема неожиданно изменилась.

Антипаттерн: Король-поставщик

Некоторые крупные компании покупают ПО для планирования ресурсов предприятия (ERP, enterprise resource planning), чтобы оптимизировать основные бизнес-задачи, такие как бухгалтерский учет, управление запасами и другие. Это работает, если компании готовы изменить свои бизнес-процессы и решения, чтобы настроить инструмент; кроме того, его можно использовать стратегически, если архитекторы представляют себе не только его преимущества, но и сопутствующие ограничения.

Однако часто организации возлагают на такое ПО слишком много надежд, что приводит к *антипаттерну Король-поставщик* (Vendor King), когда архитектура строится полностью вокруг продукта одного поставщика, что делает организацию патологически зависимой от этого инструмента. Приобретая ПО у поставщика, компании планируют дополнить пакет подключаемыми модулями, чтобы основные функции соответствовали потребностям бизнеса. Однако в большинстве случаев ERP-инструменты невозможно настроить так, чтобы полностью реализовать необходимое, и разработчики оказываются в ловушке ограничений инструмента, а также того факта, что он стал центром архитектурной вселенной. Другими словами, поставщик стал королем архитектуры организации, диктующим будущие решения.

Чтобы защититься от этого антипаттерна, рассматривайте все ПО как точку интеграции, даже если изначально оно предназначено для широкого круга задач. Это поможет заменить неэффективное поведение другими точками интеграции и свергнуть короля.

Помещая внешний инструмент или фреймворк в основу архитектуры, разработчики существенно ограничивают ее способность к эволюции как с технической точки зрения, так и с точки зрения бизнес-процессов. В части технологий возможности разработчиков будут ограничены решением поставщика в отношении персистентности, поддерживаемой инфраструктуры и множеством других

ограничений. В части бизнеса большие инкапсулирующие инструменты подвержены рискам, описанным в разделе «Антипаттерн: ловушка последних 10 % и Low Code/No Code» на с. 212. Инструменты оказываются просто не способны поддерживать оптимальный рабочий процесс; это побочный эффект ловушки последних 10 %. Большинство компаний в итоге уступают фреймворкам, изменяя свои процессы, а не пытаясь настроить инструмент. Чем больше компаний так делают, тем меньше между ними разница, что вполне нормально, пока эта разница не добавляет конкурентного преимущества. Компании часто выбирают альтернативный вариант, описанный в разделе «Подводный камень: персонализация продукта» на с. 224 и являющийся еще одной ловушкой.

Зачастую при работе с ERP-пакетами все заканчивается словами: «*Давайте закончим и скажем, что все получилось*». Поскольку эти системы требуют огромных затрат времени и денег, компании неохотно признают, что затея провалилась. Ни один технический директор не признает, что потратил впустую миллионы долларов, а поставщик не согласится, что за несколько лет внедрить инструмент так и не получилось. Поэтому обе стороны соглашаются остановиться и объявить об успехе, хотя большая часть обещанного функционала осталась нереализованной.

Не привязывайте архитектуру к «королю-поставщику».

Чтобы не стать жертвой антипаттерна Король-поставщик, рассматривайте продукты поставщиков как еще одну точку интеграции. Изолировать изменения в инструментах поставщиков, чтобы они не влияли на архитектуру, можно с помощью защитного слоя между точками интеграции.

Подводный камень: дырявые абстракции

> Все нетривиальные абстракции в какой-то степени дырявые.
>
> — *Джоэл Спольски (Joel Spolsky)*

Современное ПО состоит из абстракций: операционных систем, фреймворков, зависимостей и множества других частей. Мы создаем абстракции, чтобы нам не приходилось постоянно работать на самых низких уровнях. Если бы разработчики должны были переводить двоичный код с жестких дисков в текст, они бы никогда ничего не сделали! Развитие современного ПО во многом обусловлено тем, что мы смогли создавать эффективные абстракции.

Но за абстракции приходится платить, потому что ни одна абстракция не совершенна, иначе это была бы не абстракция, а реальная вещь. Как сказал Джоэл Спольски, все нетривиальные абстракции дырявы. Это проблема для разработчиков, потому что мы привыкли верить, что абстракции всегда точны, но они часто и неожиданно ломаются.

Возросшая сложность технологического стека усугубила проблему отвлечения абстракций. Рассмотрим типичный технологический стек примерно 2005 года, показанный на рис. 8.1.

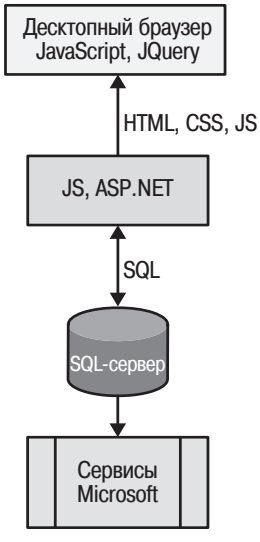


Рис. 8.1. Типичный стек технологий 2005 года

В стеке, изображенном на рис. 8.1, названия поставщиков на коробках меняются в зависимости от местных условий. Со временем, по мере роста специализации ПО, технологический стек усложняется, как показано на рис. 8.2.

Как видно на рис. 8.2, все части экосистемы расширились и приобрели дополнительную сложность. По мере усложнения проблем усложняются и их решения.

Первобытная, то есть низкоуровневая, абстракция, нарушение которой приводит к возникновению хаоса, — один из побочных эффектов усложнения технологического стека. Что, если сбой, например, неожиданный побочный эффект от простого, на первый взгляд, обращения к базе данных, возникнет в одной из низкоуровневых абстракций? Поскольку уровней много, сбой будет распространяться до вершины стека, проявляясь в пользовательском интерфейсе в виде сообщения об ошибке на глубоком уровне. По мере усложнения технологического стека становится тяжелее проводить отладку и ретроспективный анализ.

> Проанализируйте абстракцию хотя бы на один уровень ниже того, на котором вы обычно работаете.
>
> — *Коллективное мнение мудрецов разработки*

Хотя совет проанализировать нижележащий слой хорош, следовать ему становится все труднее, поскольку повышается специализация программ и, следовательно, увеличивается их сложность.

Увеличение сложности технологического стека — пример проблемы динамического равновесия. Меняется не только экосистема, но и ее компоненты, которые со временем усложняются и теснее связываются между собой. Наш механизм защиты эволюционных изменений — фитнес-функции — способен защитить хрупкие точки сцепления архитектуры. Чтобы избежать нежелательных утечек абстракции,

архитекторы задают инварианты в ключевых точках интеграции в виде фитнес-функций, которые запускаются в рамках пайплайна развертывания.

Определите уязвимые места в сложном технологическом стеке и автоматизируйте их защиту с помощью фитнес-функций.

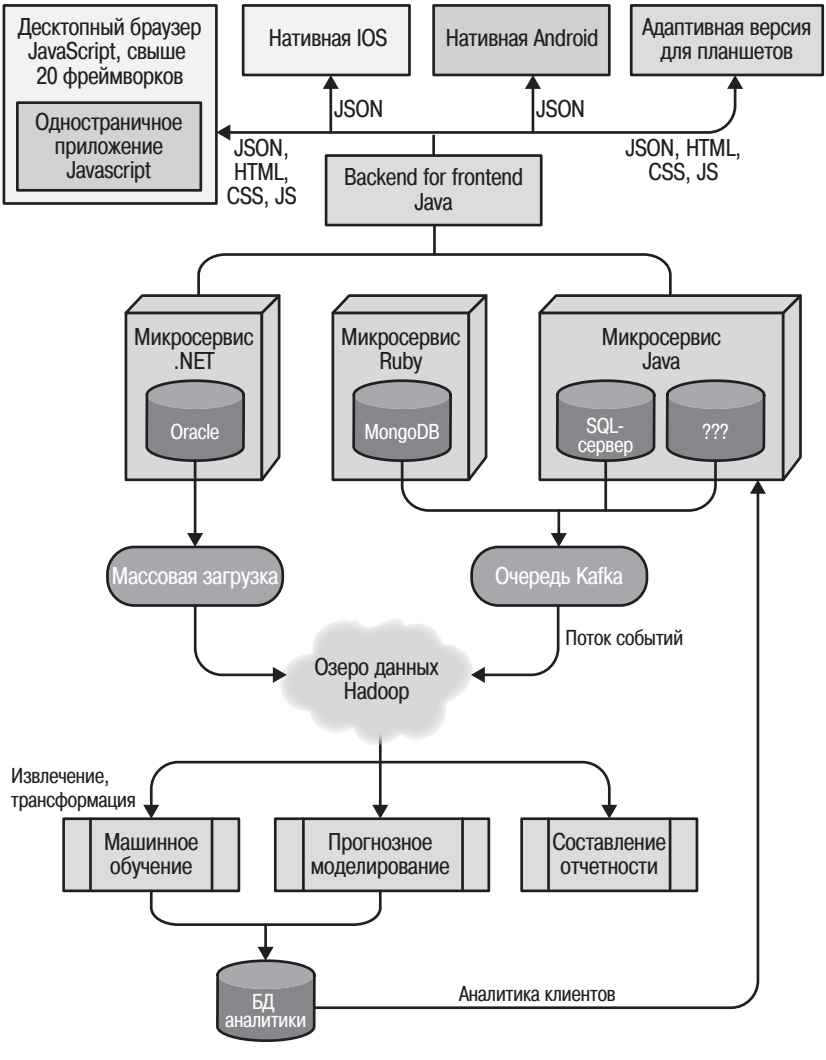


Рис. 8.2. Типичный стек программного обеспечения последнего десятилетия с большим количеством движущихся частей

Подводный камень: разработка ради строчки в резюме

Зачастую архитекторы увлекаются последними возможностями разработки и стремятся использовать все новейшие игрушки. Однако, чтобы создавать эффективную архитектуру, необходимо внимательно изучить предметную область и выбрать наиболее подходящий дизайн, реализующий максимум возможностей при минимуме разрушительных ограничений. Если, конечно, целью архитектора не является *разработка ради строчки в резюме* — использовать все возможные фреймворки и библиотеки только для того, чтобы похвастаться своими знаниями.

Не стройте архитектуру ради архитектуры; ваша цель — решить проблему.

Всегда определяйте предметную область до выбора архитектуры, а не наоборот.

Инкрементные изменения

Вносить изменения инкрементно при разработке зачастую довольно сложно. Десятилетиями программы писались с учетом в первую очередь таких факторов, как снижение затрат, совместное использование ресурсов и других внешних ограничений, а гибкость отходила на второй план. Вследствие этого во многих компаниях отсутствуют структурные элементы эволюционных архитектур.

Как отмечается в книге «Continuous Delivery»,[1] многие современные практики разработки поддерживают эволюционную архитектуру.

Антипаттерн: ненадлежащее управление

Программная архитектура не существует в вакууме — она является отражением внешних условий. Еще 10 лет назад операционные системы стоили дорого. Такими же коммерческими и дорогими были серверы баз данных, серверы приложений и вся инфраструктура для размещения приложений. Поэтому создаваемые архитектуры были нацелены на максимальное совместное использование ресурсов. Это была эпоха расцвета таких паттернов, как COA. В этой среде развивалась модель управления, поощряющая общие ресурсы как метод экономии затрат.

[1] *Хамбл Дж., Фарли Д.* «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий».

Она стала основой коммерческой мотивации для многих инструментов, таких как серверы приложений. Однако размещать множество ресурсов на машинах нежелательно из-за возникновения непреднамеренной связанности. Совместно используемые ресурсы могут быть эффективно изолированы друг от друга, но в конечном счете борьба за ресурсы дает о себе знать.

За последнее десятилетие динамическое равновесие экосистемы разработки изменилось. Теперь можно создавать архитектуры, в которых компоненты имеют высокую степень изоляции (например, микросервисы), что позволяет избежать непреднамеренной связанности в общих средах. Тем не менее многие компании по-прежнему придерживаются старой модели управления. Однако модель с упором на совместно используемые ресурсы и однородные среды становится неэффективной в современных условиях, в частности с развитием DevOps.

> Теперь все компании — разработчики.
>
> *— Журнал Forbes, 30 ноября 2011 г.*

Смысл этой известной фразы *Forbes* в том, что если приложение авиакомпании, созданное для iPad, работает плохо, то это скажется на ее итоговых показателях. Любая передовая компания, и все чаще любая компания, которая хочет оставаться конкурентоспособной, должна обладать достаточными компетециями в области разработки. И частью этой компетенции является управление активами разработки, такими как среда.

Когда появляется возможность создавать такие ресурсы, как виртуальные машины и контейнеры, без каких-либо затрат (денег или времени), модель управления с упором на одно решение становится *ненадлежащей (негодной)*. Создание множественных сред микросервисов — более эффективный подход. Одна из особенностей микросервисных архитектур — принятие полиглотных сред, когда каждая команда может выбрать подходящий стек технологий для реализации своего сервиса, а не вынуждена использовать корпоративный стандарт. Архитекторов систем предприятий, использующих традиционный подход, коробят такие слова, поскольку они абсолютно противоречат их принципам. Однако цель большинства проектов микросервисов не в том, чтобы беспорядочно перебирать технологии, а в том, чтобы подобрать именно ту, которая подходит для решения проблемы.

В современных средах использовать единый технологический стек нецелесообразно. Это приводит к непреднамеренному усложнению, когда управленческие решения только добавляют сложности и не способствуют решению задачи. Например, стандартная практика в крупных компаниях — использование реляционной базы данных одного поставщика, и на то есть очевидные причины: согласованность проектов, легкая сменяемость сотрудников и др. Однако у такого подхода есть побочный эффект: большинство проектов страдают от

избыточного проектирования. В монолитных архитектурах управленческие решения затрагивают все компоненты. Таким образом, архитектор должен выбрать базу данных, удовлетворяющую самым сложным из существующих требований каждого проекта, в котором она будет использована. К сожалению, во многих проектах такая сложность не нужна. В небольших проектах простые требования к персистентности, однако для обеспечения согласованности в них приходится использовать всю сложность промышленного сервера баз данных.

В микросервисах каждая команда может выбрать подходящий уровень сложности для своего сервиса, поскольку ни один из них не связан ни с технической архитектурой, ни с архитектурой данных. В конечном счете сложность стека сервисов приводится в соответствие с техническими требованиями. Такое разделение, как правило, наиболее эффективно, когда команда полностью владеет своим сервисом, отвечая в том числе за эксплуатационные параметры.

ПРИНУДИТЕЛЬНОЕ РАЗДЕЛЕНИЕ

Одна из особенностей архитектуры микросервисов — экстремальное техническое разделение (decoupling), позволяющее заменять сервисы без побочных эффектов. Однако при использовании одной и той же кодовой базы или даже платформы *отсутствие* связанности требует от разработчиков определенной дисциплины (поскольку соблазн повторно использовать существующий код очень велик) и мер предосторожности, чтобы не создать случайную связанность. Создание сервисов в разных технологических стеках — один из способов разделения технической архитектуры. Многие компании избегают этого подхода, поскольку опасаются, что это затруднит перемещение сотрудников между проектами. Однако Чад Фаулер (http://chadfowler.com/), бывший архитектор компании Wunderlist, придерживается противоположного мнения: он *настаивает* на использовании разных технологических стеков, чтобы избежать случайной связанности. Он считает, что случайная связанность — проблема посерьезнее, чем мобильность разработчиков.

Многие компании инкапсулируют отдельные функциональные возможности в платформу как услугу для внутреннего использования, скрывая выбор технологии (и, соответственно, возможности связанности) за хорошо определенными интерфейсами.

С практической точки зрения для крупных организаций хорошо работает модель *управления «необходимо и достаточно»*: стандартизируйте три технологических стека: простой, промежуточный и сложный — и позвольте выбирать какой-то из них на основе индивидуальных требований к разрабатываемому сервису. Это обеспечит разработчикам гибкость выбора подходящего технологического стека и в то же время сохранит для компании преимущества стандартизации.

Практический пример: «необходимое и достаточное» управление в PenultimateWidgets

В течение нескольких лет архитекторы PenultimateWidgets пытались стандартизировать всю разработку на Java и Oracle. Однако с повышением уровня детализации сервисов они поняли, что этот стек накладывает большую сложность на небольшие сервисы. Но архитекторы не готовы были полностью переходить на подход «каждому проекту свой технологический стек», поскольку им все еще была важна переносимость знаний и навыков между проектами. В итоге они выбрали «необходимое и достаточное» управление с тремя технологическими стеками:

Малый

Для очень простых проектов без жестких требований к масштабируемости или производительности они выбрали Ruby on Rails и MySQL.

Средний

Для средних проектов выбрали GoLang, а в качестве бэкенда Cassandra, MongoDB или MySQL в зависимости от требований к данным.

Крупный

Для крупных проектов остановились на Java и Oracle, поскольку они хорошо работают с различными архитектурными сложностями.

Подводный камень: недостаточная скорость выпуска релизов

Практика непрерывной доставки (http://continuousdelivery.com/) устраняет факторы, замедляющие выпуск релизов ПО, и считается аксиомой для создания успешной эволюционной архитектуры. Хотя экстремальная разновидность непрерывной доставки — непрерывное развертывание — для эволюционной архитектуры не требуется, возможность эволюции дизайна выпускаемого продукта сильно зависит от способности команд своевременно выпускать релизы.

Если инженерная культура компании основана на непрерывном развертывании, когда изменения попадают в продакшен только после пайплайна развертывания, разработчики привыкают к тому, что изменения должны быть постоянными. С другой стороны, если выпуск релизов в компании представляет собой формальный процесс, требующий большого количества специальных действий, шансы на создание эволюционной архитектуры уменьшаются.

Непрерывная доставка стремится получать результаты на основе данных и использовать метрики для оптимизации проектов. Разработчики должны уметь

измерять параметры, чтобы понять, как сделать их лучше. Одна из ключевых метрик непрерывной доставки — *время цикла* (cycle time). Это метрика, связанная с *временем производства* (lead time): время от принятия идеи до ее воплощения в работающем продукте. Однако время производства включает в себя множество субъективных действий, таких как оценка, определение приоритетов и другие, что делает его непригодной в инженерии метрикой. Вместо этого в непрерывной доставке используется *время цикла*: время от начала до завершения единицы работы, которой в данном случае является разработка продукта. Отсчет времени цикла стартует, когда разработчик начинает работать над новой функцией, и заканчивается, когда эта функция запускается на продакшен. Цель времени цикла — измерить эффективность разработки; сокращение времени цикла — одна из ключевых целей непрерывной доставки.

Время цикла важно и для эволюционной архитектуры. В биологии генетические эксперименты обычно проводятся на плодовых мушках отчасти потому, что у них быстрый жизненный цикл — новые поколения появляются достаточно быстро, чтобы увидеть ощутимые результаты. То же справедливо и для эволюционной архитектуры — более быстрое время цикла означает, что архитектура может развиваться быстрее. Таким образом, время цикла проекта определяет, насколько быстро может развиваться архитектура. Другими словами, скорость эволюции пропорциональна времени цикла, что выражается следующей формулой:

$$v \propto c,$$

где v — скорость изменений, а c — время цикла. Система не может развиваться быстрее, чем время цикла проекта. Другими словами, чем быстрее команды выпускают релизы, тем быстрее они развивают части своей системы.

Поэтому время цикла — критическая метрика эволюционной архитектуры: более быстрое время цикла подразумевает более быструю способность к эволюции. Фактически время цикла — идеальная атомарная фитнес-функция, основанная на процессе. Например, разработчики создают проект с автоматическим пайплайном развертывания, достигая времени цикла в три часа. Со временем оно постепенно увеличивается, поскольку в пайплайн добавляют все больше проверок и точек интеграции. Поскольку метрика времени выхода на рынок (time to market) важна в этом проекте, разработчики задают фитнес-функцию, которая будет подавать сигнал тревоги, если время цикла превысит четыре часа. Когда оно достигнет этого порога, можно либо перестроить работу пайплайна развертывания, либо оставить четырехчасовое значение как приемлемое. Фитнес-функции можно привязать к любому поведению, которое важно отслеживать в проектах, включая метрики проекта. Объединение важных параметров проекта в фитнес-функции позволяет разработчикам определять будущие точки принятия решений, также известные как *последний ответственный момент*.

В предыдущем примере разработчики теперь должны решить, что важнее: трехчасовое время цикла или набор тестов, которые у них есть. В большинстве проектов разработчики принимают это решение неявно, не замечая, как постепенно увеличивается время цикла, и таким образом не расставляя приоритеты между противоречащими друг другу целями. С помощью фитнес-функций можно установить пороговые значения для предполагаемых будущих точек принятия решений.

Скорость эволюции отпределяется временем цикла; более быстрое время цикла позволяет эволюционировать быстрее.

Для создания успешной эволюционной архитектуры чрезвычайно важна правильная организация проектирования, развертывания и выпуска релизов. В свою очередь, эволюционная архитектура позволяет создавать новые возможности для бизнеса с помощью разработки на основе гипотез (hypothesis-driven development).

Решение задач бизнеса

В заключение поговорим о нежелательной связанности, возникающей в связи с задачами бизнеса. В большинстве случаев бизнес не желает из вредности усложнять жизнь разработчикам; скорее неправильные с архитектурной точки зрения решения определяются его потребностями, что ведет к непреднамеренному ограничению будущих возможностей. Мы рассмотрим несколько подводных камней и антипаттернов, касающихся бизнеса.

Подводный камень: персонализация продукта

Продавцам нужно продавать. О продавцах сложилось такое представление, что они продадут любую запрашиваемую функцию, даже не зная, имеется ли она в их продукте. Таким образом, продавцам требуется продукт, который можно настроить как угодно. Однако за эту возможность приходится дорого платить.

Уникальная сборка для каждого клиента

В этом сценарии продавцы обещают уникальные версии функций в сжатые сроки, для чего разработчикам приходится использовать такие методы, как ветви контроля версий и метки для отслеживания версий.

Постоянные переключатели функций

Переключатели функций, о которых мы рассказывали в главе 3, иногда используются для задания индивидуальных настроек. Разработчики могут использовать переключатели функций для создания разных версий для разных клиентов или для создания freemium-версии продукта — бесплатной версии, в которой можно разблокировать премиум-функции за определенную плату.

Персонализация, ориентированная на продукт

В некоторых продуктах существует продвинутая возможность настройки через пользовательский интерфейс. В этом случае функции настройки являются постоянной частью приложения и требуют такого же внимания, как остальные функции продукта.

При использовании переключателей функций и персональных настроек значительно возрастает нагрузка на тестирование, поскольку в продукте появляется множество комбинаций возможных путей. Кроме увеличения числа сценариев тестирования увеличивается и количество фитнес-функций, которые необходимо задать для защиты возможных пермутаций.

Персонализация не способствует эволюционности, но это не должно быть препятствием для создания настраиваемого ПО; необходимо только реально оценить связанные с этим затраты.

Антипаттерн: система отчетов поверх системы регистрации

Назначение большинства приложений различается в зависимости от бизнес-функции. Например, одним пользователям требуется ввод заказов, а другим — отчетность для анализа. Все потребности организаций (например, ввод заказов и ежемесячную отчетность) обеспечить трудно, особенно если в основе продукта лежит монолитная архитектура и/или структура базы данных. В эпоху сервис-ориентированной архитектуры архитекторы пытались поддерживать каждую бизнес-задачу с помощью одного и того же набора «многоразовых» сервисов. Они обнаружили, что чем более универсальным был сервис, тем больше его требовалось настраивать, чтобы он стал полезен.

Система отчетов — хороший пример непреднамеренной связанности в монолитных архитектурах. Архитекторы и администраторы БД хотят использовать одну схему базы данных как для системы регистрации, так и для системы отчетов, но сталкиваются с проблемами, потому что дизайн обоих вариантов не оптимизирован ни для одного из них. Противоречие между задачами иллюстрирует распространенный подводный камень многоуровневой архитектуры. Архитекторы строят многоуровневую архитектуру, чтобы сократить случайную

связанность, создавая слои изоляции и разделения функций. Но отдельные слои нужны только для данных, а не для функции отчетности. Кроме того, маршрутизация запросов по разным уровням увеличивает задержку. Поэтому многие компании, в которых выстроена хорошая многоуровневая архитектура, позволяют разработчикам систем отчетов связывать отчеты непосредственно со схемами баз данных. Это ведет к тому, что вносить изменения в схему БД, не затрагивая отчетов, становится невозможно. Это хороший пример того, как противоречащие друг другу бизнес-цели значительно затрудняют работу архитекторов и внедрение эволюционных изменений. Никто целенаправленно не стремился к тому, чтобы развивать систему было сложно, но к этому привела совокупность принятых решений.

Во многих микросервисных архитектурах задача составления отчетности решается путем разделения поведений, когда изоляция сервисов служит для разделения, а не для консолидации. Такие архитектуры обычно строятся с использованием потоков событий или очередей сообщений для заполнения баз данных предметной области в виде «системы записей». Каждая такая БД встраивается в архитектурный квант сервиса с использованием согласованности в конечном счете, а не транзакций. Набор сервисов отчетности также прослушивает поток событий, заполняя денормализованную базу данных, оптимизированную для составления отчетов. Согласованность в конечном счете освобождает архитекторов от необходимости координации, с архитектурной точки зрения представляющей собой форму связанности, и позволяет использовать различные абстракции для различных применений приложения.

О более современном подходе к составлению отчетов на основе конкретных и аналитических данных см. в разделе «Data Mesh: ортогональная связанность данных» на с. 150 (глава 5).

Подводный камень: чрезмерно широкий горизонт планирования

Необходимость составления бюджетов и планирования часто вынуждает делать допущения и принимать ранние решения на их основе. Однако чем шире горизонт планирования без возможности пересмотреть план, тем больше решений (или допущений) принимается на основе минимального количества информации. На ранних этапах планирования разработчики тратят значительные усилия на исследования, часто представляющие собой чтение источников, чтобы подтвердить свои предположения. Определенные в ходе этих исследований «лучшие практики» или «лучшие в своем классе методы» становятся основой предположений, исходя из которых разработчики пишут код или выпускают продукт для конечных пользователей. Поскольку в эти предположения — даже если через шесть месяцев окажется, что они неверные, — вкладывается все

больше и больше усилий, от них становится все сложнее отказаться. Ловушка невозвратных затрат (https://en.wikipedia.org/wiki/Sunk_costs) касается решений, в которые были вложены эмоции. Проще говоря, чем больше времени или усилий человек вкладывает во что-то, тем сложнее от этого отказаться. В разработке это проявляется в форме *иррациональной привязанности к артефактам* — чем больше времени и усилий вы вкладываете в планирование или составление документа, тем с большей вероятностью вы будете защищать то, что содержится в плане или документе, даже осознавая, что он неточен или устарел.

Не привязывайтесь к создаваемым вручную артефактам.

Остерегайтесь длительных циклов планирования, которые вынуждают принимать необратимые решения, и ищите способы сохранить возможность выбора. Проверить применимость архитектурных решений и инфраструктуры разработки в целом можно путем разбиения больших задач на более мелкие, а также быстрой поставки. Необходимо избегать технологий, требующих значительных инвестиций еще до того, как будет создан продукт (например, крупных лицензий и контрактов на поддержку), и до того, как отзывы конечных пользователей подтвердят, что выбранная технология действительно подходит для решения проблемы.

Итоги

Как и любая архитектурная практика, эволюционная архитектура подразумевает множество компромиссов: технических, эксплуатационных, компромиссов бизнеса, данных, интеграции и т. д. Паттерны (и антипаттерны) в архитектуре наблюдаются так часто, потому что они не только дают подсказки, но и — что очень важно — определяют *контекст*, в котором эти подсказки имеют смысл. Повторное использование программных активов — очевидная цель создания архитектуры, но необходимо оценить, какие компромиссы оно может повлечь за собой: часто слишком сильная связанность оказывается хуже, чем дублирование.

Мы говорим о паттернах, но не о *лучших практиках*, которых в программной архитектуре практически не существует. *Лучшая практика* подразумевает, что всякий раз, когда архитектор сталкивается с определенной ситуацией, он может действовать автоматически по шаблону — ведь это *лучший* способ решить задачу. Однако программная архитектура полностью состоит из компромиссов, а это означает, что практически для каждого решения компромиссы необходимо оценивать заново. Паттерны и антипаттерны помогают определить контекст и то, каких антипаттернов следует избегать.

ГЛАВА 9

Эволюционная архитектура на практике

В завершение мы рассмотрим, какие шаги необходимы для реализации эволюционной архитектуры как в технологическом аспекте, так и в части бизнеса, включая организационные вопросы и вопросы работы команд. Мы также разберемся, с чего стоит начать и как грамотно представить эти шаги бизнесу.

Организационные факторы

Программная архитектура удивительным образом оказывает влияние на множество факторов, традиционно не связываемых с разработкой ПО, включая работу команд, планирование бюджета и другие. Рассмотрим, какие факторы определяют практическую применимость эволюционной архитектуры.

Не боритесь с законом Конвея

В апреле 1968 года Мелвин Конвей (Melvin Conway) представил в *Harvard Business Review* статью под названием «How Do Committees Invent?». В ней Конвей высказал идею о том, что социальные структуры, особенно взаимодействия между людьми, неизбежно влияют на конечный дизайн продукта.

Как описывает Конвей, на самой ранней стадии проектирования составляется высокоуровневое представление о системе, чтобы разделить зоны ответственности на отдельные паттерны. Характером этого разделения будут обуславливаться все возможные дальнейшие шаги команды.

Он сформулировал мысль, которая стала известна как *закон Конвея*:

> Организации, проектирующие системы <...> создают проекты, которые копируют структуру коммуникаций в этих организациях.
>
> — *Мелвин Конвей*

Как отмечает Конвей, разделение проблем на более мелкие части, которые можно делегировать, затрудняет координацию. Во многих организациях формальные структуры или жесткая иерархия, казалось бы, решают эту проблему, но эти решения негибкие. Например, в многоуровневой архитектуре, где команды сформированы в соответствии с функциями, которые они разрабатывают (пользовательский интерфейс, бизнес-логика и т. д.), решение сквозных проблем, затрагивающих все слои, увеличивает накладные расходы на координацию. Люди, перешедшие из стартапов в крупные транснациональные корпорации, скорее всего, заметят контраст между гибкой, адаптируемой культурой первых и негибкими структурами вторых. Хорошим примером закона Конвея в действии может служить попытка изменить контракт между двумя сервисами, которую трудно реализовать, если для успешного изменения сервиса, которым владеет одна команда, требуются скоординированные и согласованные усилия другой.

В своей статье Конвей фактически предупреждал архитекторов о необходимости уделять внимание не только архитектуре и дизайну ПО, но и делегированию, распределению и координации работы между командами.

Во многих организациях команды формируются в соответствии с их функциональными навыками. Например:

Фронтенд-разработчики

Команда со специализированными навыками в определенной технологии пользовательского интерфейса (HTML, мобильного или десктопного).

Бэкенд-разработчики

Команда с уникальными навыками в создании бэкенд-сервисов, иногда уровней API.

Разработчики баз данных

Команда с уникальными навыками в области создания хранилищ и логических сервисов.

Рассмотрим общую структуру/принципы формирования команд на рис. 9.1.

Такой способ организации команд относительно эффективен, если компании используют многоуровневую архитектуру, уровни в которой соответствуют представленным в разделе «Не боритесь с законом Конвея» на с. 228. Однако если команда переходит на распределенную архитектуру, такую как микросервисы, сохраняя при этом прежнюю организацию, возникнет побочный эффект увеличения объема сообщений между уровнями, как показано на рис. 9.2.

Рис. 9.1. В многоуровневых архитектурах проще разделить членов команды на группы в соответствии с их техническими возможностями

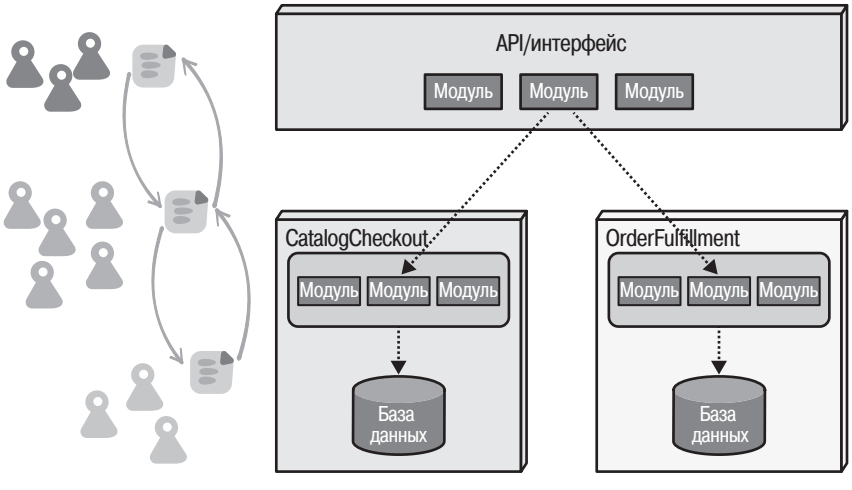


Рис. 9.2. Создание микросервисов при сохранении уровней увеличивает накладные расходы на взаимодействие

На рис. 9.2 изменения в концепциях предметной области, таких как `CatalogCheckout`, требуют координации между всеми техническими модулями, что увеличивает накладные расходы и замедляет разработку.

Функциональное разделение удобно отделу персонала, а эффективность разработки при этом снижается. Хотя каждая команда может хорошо выполнять свои задачи (например, дизайн пользовательского интерфейса, добавление

внутреннего API или сервиса либо создание нового механизма хранения данных), выпуск новой бизнес-возможности или функции требует участия всех трех команд. Команды, как правило, оптимизируют эффективность для выполнения своих непосредственных задач, а не для достижения абстрактных стратегических целей бизнеса, особенно в условиях дефицита времени. Вместо создания комплексной ценности функции команды ограничиваются созданием отдельных компонентов, совместимых или не совместимых друг с другом.

Как отмечает Конвей, *при делегировании полномочий и сужении области исследования уменьшается количество вариантов дизайна, которые можно эффективно реализовать*. Иначе говоря, человеку трудно вносить изменения, если он не владеет тем, что хочет изменить. Архитекторы должны внимательно относиться к распределению и делегированию задач, чтобы цели архитектуры соотносились со структурой команды.

Многие компании, создающие такие архитектуры, как микросервисы, формируют команды в соответствии с границами сервисов, а не с разделами технической архитектуры. В ThoughtWorks Technology Radar (https://www.thoughtworks.com/radar) мы называем это обратным законом Конвея (Inverse Conway Maneuver)[1]. Такая организация команд является идеальной, поскольку структура команды влияет на множество аспектов разработки программного обеспечения и должна отражать размер и масштаб проблемы. Например, команды, разрабатывающие архитектуру микросервисов, обычно сформированы по ее подобию, без функциональной разрозненности, и охватывают все области бизнеса и технические аспекты архитектуры. Пример командной структуры, моделирующей архитектуру, показан на рис. 9.3.

Разделение команд по архитектурному принципу встречается все чаще, поскольку все больше компаний осознают преимущества соответствия команд создаваемой архитектуре.

Структурируйте команды так, чтобы они были похожи на целевую архитектуру, и тогда вам будет легче ее создать.

В командах, сформированных на основе предметной области, а не технических возможностей, проще разрабатывать эволюционную архитектуру; кроме того, они обладают некоторыми общими характеристиками.

[1] https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

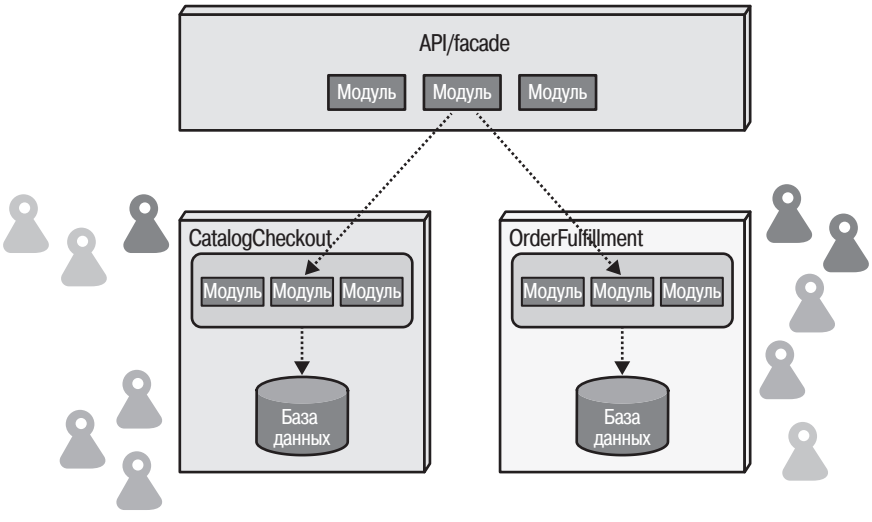


Рис. 9.3. Использование обратного закона Конвея для упрощения взаимодействия

Кроссфункциональность по умолчанию

Предметно-ориентированные команды, как правило, являются *кроссфункциональными*, то есть за каждую роль продукта в команде отвечает какой-то из ее членов. Предметная ориентированность команд позволяет избежать эксплуатационных проблем. Другими словами, команды включают все роли, необходимые для разработки, внедрения и развертывания сервиса, в том числе традиционно отдельные роли, такие как специалисты по эксплуатации. Но эти роли должны соответствовать новой структуре.

Архитекторы

Занимаются проектированием архитектуры, предполагающей отсутствие избыточной связанности, которая затрудняет инкрементные изменения. Обратите внимание, что для этого не требуются нестандартные методы, такие как микросервисы. Грамотно спроектированное модульное монолитное приложение может быть так же приспособлено к инкрементным изменениям (хотя оно должно явно поддерживать такой уровень изменений).

Бизнес-аналитики

При разработке продуктов со сложной предметной областью, обусловленной сложными правилами, конфигурацией или историей продукта, бизнес-аналитики (BA) оказывают экспертную поддержку остальным членам команды. В целях более быстрого получения обратной связи по предлагаемым изме-

нениям специалисты по бизнес-аналитике теперь непосредственно входят в кроссфункциональные команды.

Специалисты в области данных

Администраторы баз данных, аналитики данных и специалисты по data science должны решать новые проблемы, связанные с гранулярностью, транзакциями и системой регистрации данных.

Разработчики

В полностью кроссфункциональных командах, работающих со сложным технологическическим стеком, от разработчиков требуется владение разными навыками, кроме основного, или они должны быть разработчиками полного цикла и уметь работать в областях, с которыми они не сталкивались бы в изолированных подразделениях. Например, бэкенд-разработчики могут также заниматься мобильной или веб-разработкой, или наоборот.

Дизайнеры

Дизайнеры в основном работают над пользовательскими функциями в своих кроссфункциональных командах, но, возможно, им придется сотрудничать и с дизайнерами из других команд, участвующих в разработке того же продукта, чтобы обеспечить согласованность пользовательского интерфейса.

Группа эксплуатации

Нарезка сервисов и их развертывание по отдельности (часто непрерывное и параллельно с существующими сервисами) — сложная задача для многих организаций с традиционной структурой ИТ-подразделения. Простодушные архитекторы старой школы считают, что компонентная и эксплуатационная модульность — это одно и то же, но в реальном мире это не так. Автоматизация задач DevOps, таких как выделение машин и развертывание, имеет решающее значение для успеха проекта.

Продакт-менеджеры

Часто их называют генеральными директорами продукта. Обычно продакт-менеджеры (PM) определяют приоритеты потребностей клиентов и бизнес-результаты в какой-либо области, например регистрацию клиентов, платежи или поддержку клиентов. При кросс-функциональной структуре PM больше не нужно координировать работу с многочисленными техническими командами, поскольку у тех есть все навыки, необходимые для реализации своей области продукта. Работа в кроссфункциональных командах позволяет PM уделять больше времени координированию действий с другими руководителями или внутренними заинтересованными сторонами, чтобы получить бесшовный конечный продукт.

Тестировщики

Тестировщики должны решать непростые задачи обеспечения интеграционного тестирования в разных предметных областях, такие как создание интеграционных сред, установление и поддержка контрактов и т. д.

Одна из задач кроссфункциональных команд — устранение проблем координации. В традиционных изолированных командах разработчикам часто приходится ждать, пока администратор БД внесет изменения или пока кто-то из группы эксплуатации предоставит ресурсы. Локализация ролей позволит устранить сложности, возникающие при координации изолированных подразделений.

Хотя все компании мечтают, чтобы в их командах были только квалифицированные инженеры, для большинства это так и остается мечтой. Возможности найма сотрудников на ключевые роли всегда ограничены внешними обстоятельствами, такими как спрос на рынке труда. Многие компании стремятся создавать кроссфункциональные команды, но не могут этого сделать из-за нехватки специалистов. В таких случаях ограниченные ресурсы можно распределить между проектами. Например, один инженер по эксплуатации может работать в нескольких командах, а не заниматься только одним сервисом.

Благодаря организации архитектуры и команд вокруг предметной области, изменения теперь обрабатываются в одной команде, что уменьшает степень трения. Архитектура, ориентированная на предметную область, может по-прежнему использовать преимущества многоуровневой архитектуры, такие как разделение функций. Например, реализация конкретного микросервиса может зависеть от фреймворка, реализующего многоуровневую архитектуру, что позволяет легко переходить на другой технический уровень. Микросервисы инкапсулируют техническую архитектуру внутри предметной области и меняют порядок традиционного взаимодействия в командах.

КОМАНДЫ «НА ДВЕ ПИЦЦЫ» В AMAZON

Компания Amazon прославилась своим подходом к командной работе, который она называла *команды на две пиццы*. Его суть в том, что в команде должно быть столько людей, сколько можно накормить двумя большими пиццами, и не больше. Такое ограничение исходит скорее из эффективности коммуникации, чем из размера команды, — чем больше команда, тем с большим количеством людей приходится взаимодействовать каждому ее члену. Каждая команда в компании является кроссфункциональной и придерживается философии «ты создаешь — ты и управляешь», а это означает, что команда полностью владеет своим сервисом, включая его эксплуатацию.

Деление на небольшие кроссфункциональные команды также обусловлено биологическими особенностями человека. Команды Amazon «на две пиццы» имитируют поведение приматов в малых группах. В спорте большинство команд состоит из плюс-минус 10 игроков, и антропологи считают, что наши предки охотились группами примерно такого же размера. Создание команд, наделенных высокой ответственностью, активирует социальные механизмы поведения: члены таких команд становятся более ответственными. Представим, например, что разработчик в традиционной структуре два года назад написал код, который сломался посреди ночи, и кому-то из группы эксплуатации пришлось вставать по звонку и исправлять его. На следующее утро наш нерадивый разработчик может даже не понять, что случайно вызвал ночную панику. Но если бы подобное произошло в кроссфункциональной команде, то на следующее утро нашему незадачливому разработчику пришлось бы смотреть в глаза своему невыспавшемуся коллеге, которого он случайно потревожил. Это должно мотивировать небрежного сотрудника быть более ответственным.

Создание кроссфункциональных команд предотвращает ситуации, когда каждый указывает пальцем на другого, и формирует чувство сопричастности, мотивируя команду работать лучше.

ВЫСВОБОЖДЕНИЕ РЕСУРСОВ БЛАГОДАРЯ АВТОМАТИЗАЦИИ DEVOPS

Однажды Нил консультировал компанию — провайдера услуг хостинга. В ней было свыше десяти команд разработчиков, занимавшихся четко определенными модулями. Однако всем обслуживанием, обеспечением, мониторингом и другими общими задачами управляла группа эксплуатации. Менеджеру часто жаловались разработчики, которым требовалось быстрее получать необходимые ресурсы, такие как базы данных и веб-серверы. Он придумал решение, что сотрудник группы эксплуатации по одному дню в неделю будет заниматься каждым проектом. В этот день разработчики были счастливы: им не приходилось ждать! Увы, у менеджера не хватало ресурсов, чтобы сделать эту практику регулярной.

Точнее, так он думал. Мы обнаружили, что большая часть ручной работы, выполняемой группой эксплуатации, представляла собой случайную сложность: неверно сконфигурированные машины, компот из производителей и брендов и множество других проблем, которые можно решить. Настроив каталоги, мы помогли автоматизировать выделение новых машин с помощью Puppet (http://puppetlabs.com/). После этого ресурсов группы эксплуатации хватило, чтобы выделить инженера на каждый проект и при этом управлять автоматизированной инфраструктурой.

Компания не нанимала новых инженеров и не меняла их должностные обязанности. Она применила современные практики автоматизации процессов, не требующих регулярного участия людей, освободив сотрудников, чтобы они больше могли заниматься разработкой.

Организуйте команды исходя из возможностей бизнеса

Организация команд вокруг предметной области подразумевает также их организацию исходя из бизнес-возможностей. Многие организации ожидают, что их техническая архитектура будет представлять собой собственную сложную абстракцию, слабо связанную с поведением бизнеса, потому что в рамках традиционного подхода архитекторы создают чистую техническую архитектуру с разделением по функциональности. Многоуровневая архитектура призвана облегчать замену уровней технической архитектуры, а не работу над такой сущностью предметной области, как `Customer`. Это обусловлено главным образом внешними факторами. Например, многие архитектурные стили последнего десятилетия в основном сосредоточены на максимизации совместного использования ресурсов из-за их стоимости.

Архитекторы постепенно освобождаются от коммерческих ограничений, внедряя открытый исходный код во всех частях систем организаций. Архитектуре совместно используемых ресурсов присущи проблемы, связанные с непреднамеренным смешением ее частей. Теперь, когда у разработчиков есть возможность создавать клиентские среды и функциональность, им проще перейти от технических архитектур к ориентированным на предметную область, чтобы подобрать оптимальную единицу изменений для большинства программных проектов.

Организуйте команды на основе бизнес-возможностей, а не функциональности.

Соотносите когнитивную нагрузку с возможностями бизнеса

Со времени выхода первого издания этой книги в отрасли появились более совершенные подходы к формированию команд, оптимизированные для непрерывного потока создания ценности. В своей книге «Team Topologies: Organizing Business and Technology Teams for Fast Flow» Мануэль Пайс (Manuel Pais) и Мэтью Скелтон (Matthew Skelton) выделяют четыре модели команд.

Потоковые команды (Stream-aligned teams)
: работают в соответствии с потоком задач бизнес-области (или, чаще, ее сегмента).

Вспомогательные команды (Enabling teams)
: помогают потокоориентированным командам справляться со сложностями и выявляют новые возможности, такие как обучение новым навыкам/технологиям.

Команды сложных подсистем (Complicated subsystem teams)

владеют частью бизнес-области, требующей больших математических/расчетных/технических знаний.

Платформенные команды (Platform teams)

представляют собой объединение команд других типов и создают убедительный внутренний продукт для повышения скорости работы потокоориентированных команд.

В моделях этих авторов использование потокоориентированных команд соответствует нашей рекомендации по формированию команд вокруг возможностей бизнеса с небольшой оговоркой: при формировании команды необходимо также учитывать *когнитивную нагрузку*. Команда, на которой лежит чрезмерная когнитивная нагрузка, либо из-за сложной предметной области, либо из-за сложного набора технологий будет с трудом справляться с поставленной задачей. Например, если вы когда-либо работали с системой обработки платежей, то знаете, что когнитивная нагрузка, создаваемая правилами и исключениями для конкретных платежных схем, очень высока. Одна команда может работать с одной платежной схемой; но если эта команда должна поддерживать пять или шесть параллельных схем, она, скорее всего, не справится с когнитивной нагрузкой даже без учета дополнительной технической сложности.

Решением проблемы может стать создание нескольких потокоориентированных команд или, при необходимости, команды подсистемы. Например, у вас может быть одна потокоориентированная команда, которая занимается сквозным процессом обработки платежей, и команда подсистемы для поддержания конкретной схемы оплаты (например, Mastercard или Visa).

Книга «Team Topologies» поддерживает идею о том, что команды необходимо организовывать в соответствии с бизнес-возможностями, но учитывать при этом когнитивную нагрузку.

Думайте о продукте, а не о проекте

Один из механизмов, часто используемых для смещения акцентов в работе команды, — организация работы вокруг *продуктов*, а не *проектов*. В большинстве организаций для всех программных проектов рабочий процесс одинаков. Определяется задача, формируется команда разработчиков, которые работают над задачей до ее «завершения», после чего передают ПО группе эксплуатации для «ухода», «кормления» и обслуживания в течение всего жизненного цикла. Затем команда проекта переходит к следующей задаче.

Такой подход создает множество проблем. Во-первых, поскольку команда переключилась на другие задачи, исправление ошибок и другие операции

обслуживания часто сложно контролировать. Во-вторых, поскольку разработчикам неважны эксплуатационные параметры своего кода, они меньше заботятся о таких вещах, как качество. В целом чем больше уровней косвенности между разработчиком и его кодом, тем меньше у разработчика связи с этим кодом. Иногда это приводит к противостоянию подразделений, что неудивительно, поскольку многие организации поощряют конкуренцию между сотрудниками. В-третьих, понятие «проект» подразумевает что-то временное: проекты заканчиваются, что влияет на процесс принятия решений теми, кто над ними работает.

Восприятие ПО как *продукта* меняет перспективу компании в трех направлениях. Во-первых, продукты — это навсегда, в отличие от проектов. Кроссфункциональные команды (часто основанные на обратном законе Конвея) остаются связанными со своим продуктом. Во-вторых, у каждого продукта есть владелец, который защищает его использование в экосистеме и управляет такими параметрами, как требования. В-третьих, поскольку команда является кроссфункциональной, в ней представлены все роли, необходимые для поддержки продукта: менеджеры по продукту, бизнес-аналитики, дизайнеры, разработчики, QA, администраторы БД, группа эксплуатации и т. д.

Переход от менталитета *проекта* к менталитету *продукта* поможет обеспечить долгосрочную поддержку компании. Команды продукта берут на себя ответственность за его долгосрочное качество. Таким образом, разработчики отвечают за качество и уделяют больше внимания недостаткам. Смена перспективы также помогает сформировать долгосрочное видение в команде. Книга «Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework» Мика Керстена (Mik Kersten) (IT Revolution Press) рассказывает о возможных изменениях в организации и о том, как успешно провести организацию через эти культурные и структурные изменения.

Не создавайте слишком большие команды

Многие компании на собственном опыте убедились, что большие команды работают неэффективно, и Дж. Ричард Хэкман (J. Richard Hackman), известный эксперт в области динамики команд, предлагает тому объяснение. Дело не в количестве сотрудников, а в количестве связей между ними. Для определения этого количества он использует формулу, представленную в уравнении 9.1, где n — количество человек.

Уравнение 9.1. Количество связей между людьми

$$\frac{n(n-1)}{2}$$

В уравнении 9.1 по мере увеличения количества человек в команде быстро растет и количество связей между ними, как показано на рис. 9.4.

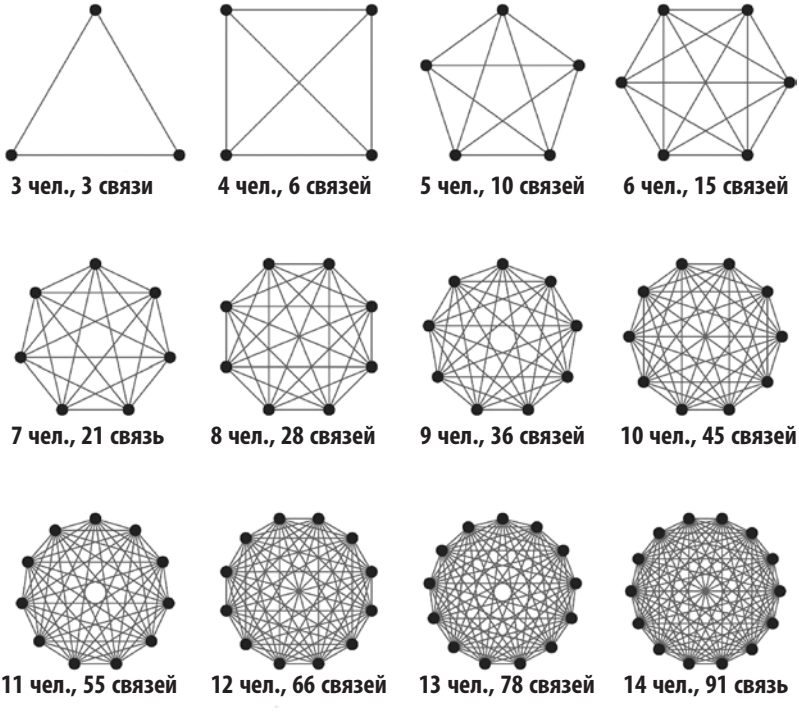


Рис. 9.4. По мере увеличения количества человек быстро растет и количество связей

На рис. 9.4 видно, что в команде из 14 человек 91 связь; а когда членов команды становится 50, количество связей становится пугающим — 1225. Таким образом, практика создания небольших команд связана со стремлением сократить объем взаимодействий. И эти малые команды должны быть кроссфункциональными, чтобы устранить искусственное трение, возникающее при координации изолированных подразделений, которая часто ведет к увеличению числа участников проекта.

Команда не должна знать, что делают другие команды, если только между командами нет точек интеграции. Даже в этом случае для обеспечения целостности точек интеграции следует использовать фитнес-функции.

Стремитесь сокращать количество связей между командами разработчиков.

Связанность команды

Способы организации и управления в компании существенно влияют на разработку программного обеспечения. В этом разделе мы рассмотрим различные вопросы организации работы компании и команд, которые способствуют или, наоборот, препятствуют созданию эволюционных архитектур. Большинство архитекторов не задумываются, как структура команды влияет на связанность архитектуры, но это влияние огромно.

Культура

> **Культура** (сущ.): принципы, обычаи и социальное поведение определенного народа или общества.
>
> — *Оксфордский словарь английского языка*[1]

Архитекторы должны следить за процессом создания систем и за поведением, поощряемым в организации. От их действий и от того, каким образом они принимают решения о выборе инструментов и создании проектов, во многом зависит, будет ли создаваемый продукт эволюционным. Хорошие архитекторы берут на себя роль лидеров, создавая техническую культуру и разрабатывая подходы к созданию систем. Они обучают инженеров и мотивируют их приобретать навыки, необходимые для построения эволюционной архитектуры.

Архитектор может понять инженерную культуру команды, задавая такие вопросы:

- Все ли в команде знают, что такое фитнес-функции, и учитывают влияние выбора новых инструментов или продуктов на способность внедрять новые фитнес-функции?
- Измеряют ли команды, насколько хорошо их система соответствует заданным фитнес-функциям?
- Понимают ли инженеры, что такое связность и связанность? А что такое коннасценция?
- Обсуждалось ли в команде, какие технические понятия и понятия, относящиеся к предметной области, связаны друг с другом?
- Выбирают ли команды решения не на основе того, какую технологию они хотят освоить, а на основе того, позволяет ли она вносить изменения?
- Как команды обрабатывают изменения бизнес-требований? Трудно ли им внедрять небольшие изменения? Тратят ли они на них слишком много времени?

[1] В Оксфордском словаре определение, разумеется, дано на английском языке. — *Примеч. ред.*

Изменение поведения команды часто требует изменений сопутствующих процессов, поскольку люди реагируют на то, что от них требуется сделать.

> Скажи мне, как ты меня оцениваешь, и я скажу тебе, как я буду себя вести.
>
> *— Д-р Элияху М. Голдратт (Dr. Eliyahu M. Goldratt)*
> *(The Haystack Syndrome[1])*

Если команда не привыкла к изменениям, архитектор может внедрить практику, поощряющую их внедрение. Например, если команда обдумывает переход на новую библиотеку или фреймворк, архитектор может попросить провести короткий эксперимент и оценить, сколько связанности это добавит. Смогут ли инженеры писать и тестировать код вне данной библиотеки или фреймворка либо потребуется дополнительно настраивать новые инструменты во время выполнения, что может замедлить цикл разработки?

Анализ кода при выборе новых библиотек или фреймворков лучше всего позволит понять, в какой степени недавно измененный код поддерживает внедрение изменений. Если в системе есть другие места, которые будут использовать другую внешнюю точку интеграции и эта точка интеграции изменится, сколько мест нужно будет обновить? Конечно, необходимо избегать перепроектирования, раннего добавления дополнительной сложности или изменения абстракций. Книга «Refactoring»[2] (https://refactoring.com/) дает следующий совет:

> Когда вы делаете что-то в первый раз, просто делайте это. Делая это во второй раз, вы морщитесь от перспективы дублирования, но все равно дублируйте. В третий раз проводите рефакторинг.

Многие команды чаще всего ставят своей целью предоставление новой функциональности и вознаграждаются именно за достижение этой цели, а качество кода и эволюционность учитываются только в том случае, если это отдельный приоритет. Архитектор, стремящийся создавать эволюционную архитектуру, должен следить, чтобы команда в первую очередь выбирала решения, способствующие эволюционности, или находить способы поощрить подобную модель.

Культура экспериментирования

Для успешной эволюции необходимы эксперименты, но некоторые компании не уделяют им достаточно времени, потому что слишком заняты выполнением планов. Успешное экспериментирование подразумевает регулярное тестирова-

[1] «Синдром стога сена. Выуживание информации из океана данных»

[2] *Фаулер М.* «Рефакторинг: улучшение проекта существующего кода».

ние (как с технической точки зрения, так и с точки зрения продукта) новых идей в небольшом объеме и интеграцию успешных экспериментов в существующие системы.

> Реальный успех измеряется количеством экспериментов, которые можно провести за 24 часа.
>
> — *Томас Алва Эдисон (Thomas Alva Edison)*

Проведение экспериментов в организациях можно поощрять разными способами.

Заимствование идей извне

Многие компании отправляют своих сотрудников на конференции и мотивируют их к поиску новых технологий, инструментов и подходов для лучшего решения проблем. Другие компании привлекают внешних советников или консультантов в качестве источников новых идей.

Поощрение явного совершенствования

Компания Toyota наиболее известна своей культурой кайдзен, или непрерывного совершенствования. Кайдзен предполагает, что каждый должен постоянно искать возможности совершенствования, особенно тот, кто непосредственно работает с проблемой и уполномочен ее решать.

Работа методом всплесков и стабилизаций

Спайк (от англ. spike — «всплеск») — это практика экстремального программирования, при которой команды генерируют случайное решение, чтобы быстро изучить сложную техническую проблему, исследовать незнакомую область или повысить уверенность в оценке. При использовании спайк-решений скорость изучения увеличивается за счет снижения качества продукта; спайк-решение не внедряется на продакшен, потому что оно недостаточно продуманно и работоспособно. Оно создается для изучения и не является хорошо спроектированным решением.

Выделение времени на инновации

Google известен своей практикой «20 %», когда сотрудники 20 % своего времени могут работать над любым проектом. Другие компании организуют хакатоны и поощряют команды находить новые продукты или улучшать существующие. Atlassian регулярно проводит 24-часовые сессии — ShipIt days (https://www.atlassian.com/company/shipit).

Внедрение разработки на основе наборов

Разработка на основе наборов предполагает исследование нескольких подходов. На первый взгляд работа с несколькими вариантами требует дополни-

тельных усилий, но после изучения нескольких вариантов команды в итоге лучше понимают поставленную задачу и обнаруживают реальные ограничения, связанные с инструментарием или подходом. Эффективная разработка на основе наборов подразумевает создание прототипов нескольких подходов за короткое время (то есть менее нескольких дней), чтобы получить более конкретные данные и опыт. Надежный продукт часто является результатом изучения нескольких конкурирующих решений.

Связь инженеров с конечными пользователями

Эксперимент можно считать успешным только тогда, когда команды понимают эффект от своей работы. Во многих компаниях с экспериментальным мышлением команды и специалисты по продукту лично оценивают влияние решений на клиентов и поощряют эксперименты для изучения этого влияния. A/B-тестирование — одна из таких практик. Другая практика, которую применяют передовые компании, — привлечение команд и инженеров к наблюдению за тем, как пользователи взаимодействуют с продуктом для выполнения определенной задачи. Эта практика юзабилити развивает эмпатию к конечным пользователям и помогает инженерам лучше понять потребности пользователей и разрабатывать новые идеи для их удовлетворения.

Финансы и планирование бюджета

Многие традиционные функции архитектуры предприятия, такие как планирование бюджета, должны соответствовать меняющимся приоритетам эволюционной архитектуры. Прежде планирование бюджета основывалось на способности предсказывать долгосрочные тенденции в экосистеме программной разработки. Однако, как мы уже отмечали, фундаментальная природа динамического равновесия затрудняет составление прогнозов.

На самом деле существует интересная взаимосвязь между квантами архитектуры и ее стоимостью. По мере увеличения количества квантов стоимость одного кванта снижается, пока не будет достигнуто оптимальное значение, как показано на рис. 9.5.

На рис. 9.5 видно, что с увеличением количества квантов архитектуры стоимость каждого из них уменьшается из-за влияния нескольких факторов. Во-первых, чем меньше составные части архитектуры, тем более дискретно и определенно разделение задач. Во-вторых, растущее количество физических квантов требует автоматизировать их эксплуатацию, поскольку после определенного момента эту работу уже нецелесообразно выполнять вручную.

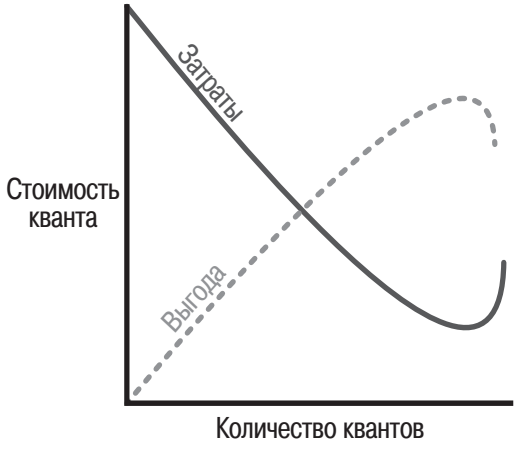


Рис. 9.5. Взаимосвязь между архитектурными квантами и стоимостью

Однако можно сделать кванты настолько мелкими, что из-за количества они станут более дорогостоящими. Например, в архитектуре микросервисов можно создавать сервисы на уровне одного поля формы. На этом уровне стоимость координации каждой маленькой части начинает доминировать над другими факторами архитектуры. Таким образом, на крайних участках графика большое количество квантов приводит к снижению выгоды на квант.

В эволюционной архитектуре соотношение между размером кванта и его стоимостью должно быть оптимальным. В каждой компании оно разное. Например, компания, работающая на агрессивном рынке, должна работать быстро и поэтому кванты в ней должны быть меньше. Помните, что скорость появления новых поколений продукта пропорциональна времени цикла, а как правило, чем меньше квант, тем меньше время цикла. Для другой же компании достаточно простой монолитной архитектуры.

Поскольку мы работаем с экосистемой, которая не поддается планированию, оптимальное соответствие между архитектурой и стоимостью зависит от многих факторов. Это подтверждает наше наблюдение о том, что роль архитекторов расширилась: выбор архитектуры сейчас более важен, чем когда-либо.

Вместо того чтобы придерживаться складывавшихся десятилетиями «лучших практик», современные архитекторы системы предприятия должны представлять себе как преимущества эволюционных систем, так и характерную для них неопределенность.

Значение для бизнеса

В этой книге мы рассматриваем множество технических деталей, но если вы не сможете обосновать ценность выбранного подхода для бизнеса, то нетехнические специалисты сочтут его метаработой. Таким образом, архитекторы должны уметь доказать, что эволюционная архитектура способна повысить доверие к изменениям и способствовать автоматизации управления. Однако более очевидные ее преимущества заключаются в возможностях, которые предоставляет этот архитектурный подход.

Чтобы продать идею эволюционной архитектуры, необходимо говорить с заинтересованными сторонами бизнеса на понятном им языке (а не в терминах архитектуры): расскажите им об A/B-тестировании и возможности изучить поведение клиентов. В основе этих продвинутых методов взаимодействия лежат вспомогательные механизмы и структуры эволюционной архитектуры, включая возможность разработки на основе гипотез и данных.

Разработка на основе гипотез и данных

Случай с GitHub, описанный в разделе «Практический пример: изменение архитектуры при развертывании 60 раз в день» на с. 105 (глава 4), касающийся использования фреймворка `Scientist`, является примером *разработки на основе данных* — когда данные инициируют изменения, касающиеся в основном технологической части. Аналогичный подход, касающийся компонентов бизнеса, называется *разработкой на основе гипотез*.

В период между Рождеством-2013 и Новым годом-2014 Facebook столкнулся с проблемой: за неделю туда было загружено больше фотографий, чем всего размещено на Flickr, и более миллиона из них были помечены как оскорбительные.

Facebook позволяет пользователям отмечать фотографии, которые они считают потенциально оскорбительными, а затем проводит собственную проверку, чтобы оценить, действительно ли они являются таковыми. Однако резкое увеличение количества фотографий вызвало нехватку сотрудников для их просмотра.

К счастью, в Facebook существуют современные подходы DevOps и возможность проводить эксперименты со своими пользователями. Когда инженера Facebook спросили, с какой вероятностью типичный пользователь Facebook будет вовлечен в какой-либо эксперимент, он ответил: «С вероятностью 100 % — у нас обычно проходят более 20 экспериментов в одно и то же время». Инженеры использовали эксперимент, чтобы задать пользователям вопрос, *почему* они посчитали фотографии оскорбительными, и услышали множество необычных причин. Например, люди не любят признавать, что они плохо выглядят на

фотографии, но охотно соглашаются, что неудачное фото — это вина фотографа. Экспериментируя с формулировками и вопросами, инженеры опросили реальных пользователей и выяснили, почему они отмечали фотографии как оскорбительные. Создав платформу, позволяющую проводить эксперименты, за относительно короткое время Facebook выявил много нерелевантных причин, что позволило решить проблему большого количества оскорбительных фотографий.

В книге «Lean Enterprise» (O'Reilly) авторы описывают современный процесс *разработки на основе гипотез*. Согласно этой методологии, следует использовать научный подход вместо того, чтобы собирать формальные требования и тратить время и ресурсы на создание функций в приложениях. Команды создают минимально жизнеспособную версию приложения (будь то новый продукт или проект обслуживания существующего приложения), а затем строят гипотезы в процессе разработки новых функций. В рамках этой методологии гипотезы формулируются с таким расчетом, чтобы проверить, какие эксперименты необходимо провести для получения результатов и каким образом подтверждение гипотезы повлиет на будущее развитие приложения.

Например, вместо того чтобы менять размер изображения товаров на странице каталога только потому, что так хочет бизнес-аналитик, сформулируйте гипотезу: мы предполагаем, что увеличение размера изображений приведет к повышению продаж товаров на 5 %. После этого проведите эксперименты с помощью A/B-тестирования — в одной группе разместите более крупные изображения товаров, а в другой стандартные — и проанализируйте результаты.

Даже в agile-проектах с вовлечением бизнес-пользователей постепенно возникают проблемы. Решение бизнес-аналитика само по себе может быть верным, но в сочетании с другими функциями в конечном итоге затруднит взаимодействие с пользователями. Отличный пример — команда mobile.de (http://mobile.de/), которая бессистемно накапливала новые функции, пока продажи не начали снижаться, и одной из причин тому стал чрезвычайно запутанный пользовательский интерфейс — частый результат продолжения работы над зрелыми программными продуктами. Чтобы справиться с проблемой, команда пробовала разные варианты: увеличение количества списков, оптимизацию расстановки приоритетов и группировки. Команда разработала три версии пользовательского интерфейса и позволила своим пользователям выбрать предпочтительную.

Движущей силой agile-методологии являются вложенные циклы обратной связи: тестирование, непрерывная интеграция, итерации и так далее. И все же та часть цикла, которая включает в себя конечных пользователей приложения, ускользает от внимания команд. Разработка на основе гипотез дает нам беспрецедентную возможность учесть параметр пользователей, изучая их поведение и на его основе создавая то, что действительно представляет для них ценность.

Разработка на основе гипотез требует координации многих изменяющихся составляющих: эволюционной архитектуры, современного DevOps, модифицированного сбора требований и способности запускать несколько версий приложения одновременно. В архитектурах на основе сервисов (например, микросервисы) одновременный запуск нескольких версий обычно реализуется благодаря интеллектуальной маршрутизации сервисов. Например, при запуске приложения какой-то пользователь может использовать одну группу сервисов, а какой-то запрос — совершенно иной набор экземпляров тех же сервисов. Если большинство сервисов содержит широкий набор запущенных экземпляров (например, в целях масштабируемости), то разумно несколько расширить функциональность каких-то из этих экземпляров и перенаправить к ним часть пользователей.

Чтобы получить значимые результаты, эксперименты должны быть достаточно длительными. Лучше найти измеримый способ определения оптимального варианта, чем раздражать пользователей всплывающими опросами. Например, позволит ли предполагаемый рабочий процесс сократить количество нажатий клавиш и щелчков мышью, которые требуется сделать пользователю для выполнения задачи? Просто по умолчанию включив пользователей в цикл обратной связи при разработке и проектировании, вы сможете создать гораздо более функциональный продукт.

Применение фитнес-функций в экспериментах

Фитнес-функции часто применяются для проверки гипотез. Архитекторам приходится выбирать решения для сценариев, которые ранее никогда не возникали, поэтому они строят обоснованные гипотезы. После реализации решения разработчиками архитектор может использовать фитнес-функции для проверки своих гипотез. Рассмотрим несколько реальных примеров.

Практический пример: реализация протокола UDP

Экосистема PenultimateWidgets включает большое количество задач ETL и пакетных процессов. Для обеспечения выполнения задач, таких как `Send Reports`, `Consolidate Information` и др., команда создала пользовательский инструмент мониторинга, показанный на рис. 9.6.

Система на рис. 9.6 спроектирована с использованием протокола UDP между задачами ETL и сервисом мониторинга. Иногда в ней терялись сообщения о завершении задач, что приводило к ложному срабатыванию оповещений, и команде приходилось выделять сотрудника для их обработки. Архитекторы решили создать фитнес-функцию, чтобы определить, какая доля сообщений не покрывается инструментом мониторинга. Если это число оказалось бы выше 10 %, инструмент должен был быть заменен на стандартную реализацию.

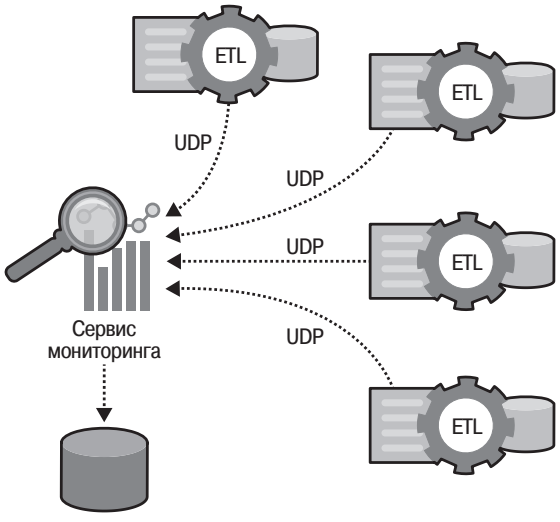


Рис. 9.6. Пользовательский инструмент мониторинга задач ETL

Чтобы проверить гипотезу о ненадежности пользовательского инструмента, команда построила фитнес-функцию, позволяющую сделать следующее:

- Рассчитать предполагаемое количество сообщений из всех приложений и частоту их получения в контролируемой среде (например, PreProd или UAT) с помощью мониторинга.
- Создать сервис имитации `Mock Service` для имитации этого количества запросов.
- Использовать `Mock Service` для чтения обработанных сообщений из базы данных `Monitor Service`, чтобы измерить процент потерянных сообщений и максимальное количество сообщений, которые приложение может обработать без сбоев, и сохранить эту информацию в JSON-файл.
- Обработать JSON-файл с помощью инструмента аналитики, например Pandas (https://pandas.pydata.org/), и вывести результат.

Схема фитнес-функции изображена на рис. 9.7.

После обработки результатов команда пришла к выводу, что при большом масштабе теряется около 40 % сообщений, поэтому надежность пользовательского решения сомнительна и требуется изменить реализацию.

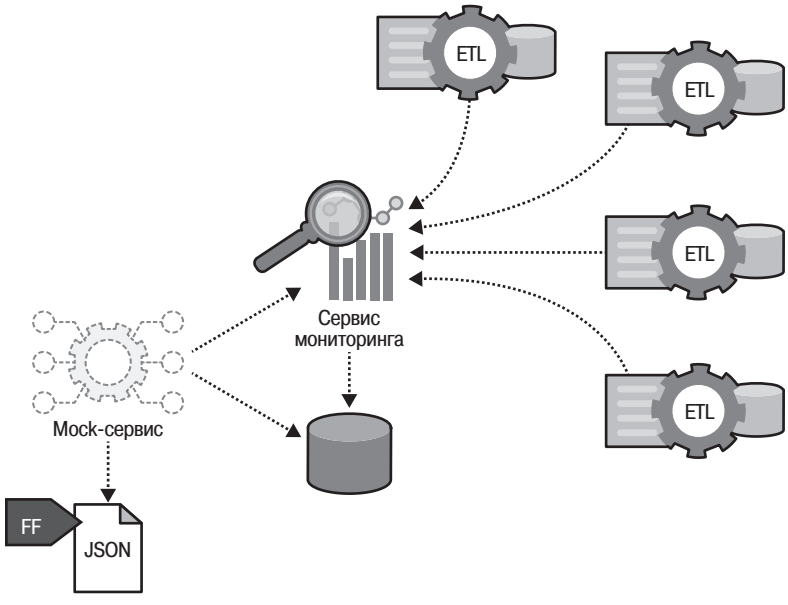


Рис. 9.7. Фитнес-функция для проверки гипотезы

Практический пример: зависимости безопасности

В PenultimateWidgets существует значительная брешь в безопасности некоторых библиотечных зависимостей, что вынудило команду осуществлять длительный процесс проверки цепочки поставок ПО вручную. Однако он значительно замедлял работу и не соответствовал требованиям рынка.

Чтобы уменьшить время обратной связи при проверках безопасности, команда создала в пайплайне непрерывной интеграции этап сканирования списка зависимостей библиотек, на котором каждая версия сверялась с обновляемым в реальном времени списком блокировок и выдавалось предупреждение, если в каком-либо проекте использовалась включенная в него библиотека, как показано на рис. 9.8.

Рис. 9.8 иллюстрирует, как можно применять фитнес-функции для целостного управления важными параметрами экосистемы. Скорость обратной связи — критическое требование к безопасности, и автоматизация проверок безопасности обеспечивает максимально быстрое ее получение. Она не заменит человека в цикле обратной связи, но позволит автоматизировать регрессию и другие задачи, чтобы люди могли заняться решением проблем, требующих творческого подхода, который под силу применить только человеку.

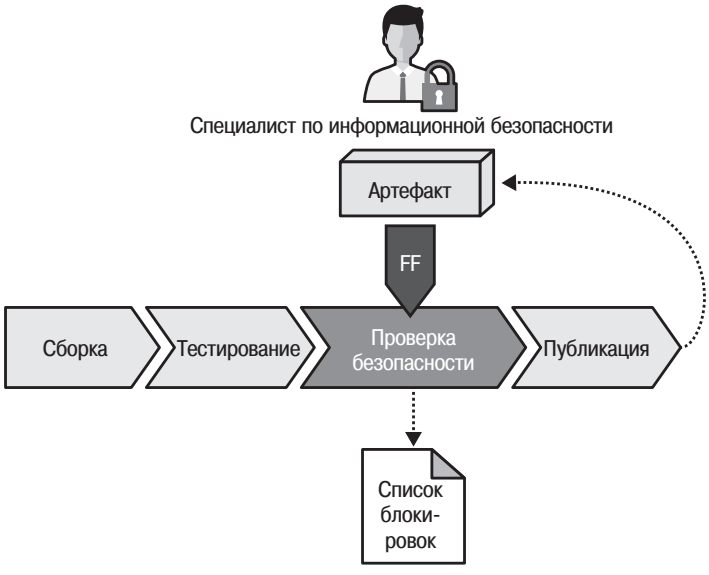


Рис. 9.8. Сканирование безопасности во время непрерывной интеграции

Практический пример: фитнес-функция параллелизма

PenultimateWidgets использовала паттерн Strangler Fig (Фикус-душитель) — постепенную замену функциональности по одному дискретному образцу поведения за раз. Для обработки выбранной части предметной области команда создала новый микросервис. Он выполняется в рабочей среде и использует двойную запись, но сохраняет первоисточник в унаследованной базе данных. Поскольку команда никогда не писала подобные сервисы, архитекторы предварительно оценили, что коэффициент масштабирования должен составлять 120 запросов в секунду. Однако сервис часто падал, несмотря на то что мог обрабатывать до 300 запросов в секунду. Необходимо было решить, увеличивать ли коэффициент автомасштабирования, или проблема была в другом — см. рис. 9.9.

Как показано на рис. 9.9, команда создала фитнес-функцию для измерения реальной производительности в рабочей среде. Эта фитнес-функция позволила сделать следующее:

1. Подсчитать количество входящих вызовов на продакшен, проверяя, какое максимальное количество запросов сервис должен быть способен обрабатывать и каков должен быть коэффициент автомасштабирования, чтобы обеспечить доступность при горизонтальном масштабировании.
2. Создать запрос New Relic, получающий количество вызовов на продакшен в секунду.

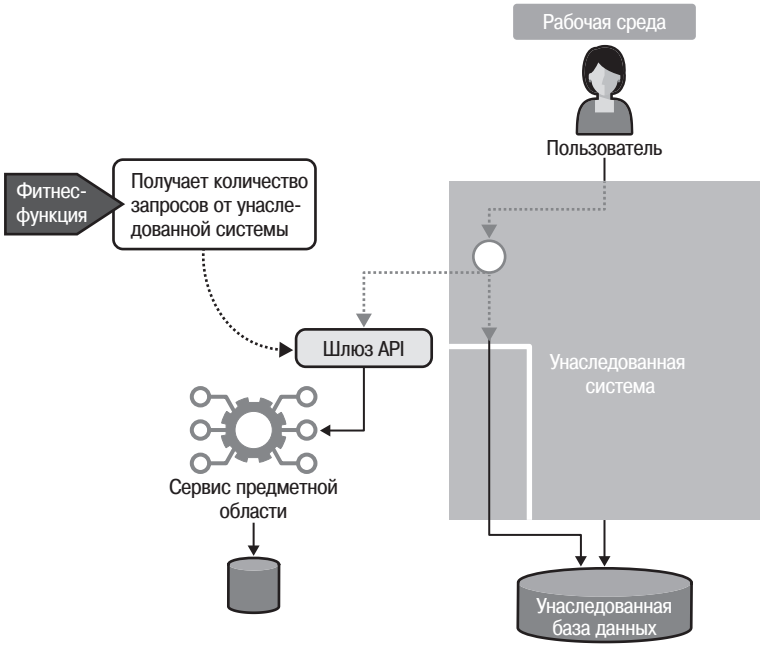


Рис. 9.9. Проверка уровней параллелизма

3. Провести новое нагрузочное тестирование параллельных потоков, используя полученное количество вызовов в секунду.
4. Отследить память и загрузку процессора и опредить точку напряжения.
5. Затем эту фитнес-функцию поместили в пайплайн, чтобы обеспечить постоянную доступность и производительность.

После выполнения фитнес-функции команда поняла, что в среднем в секунду осуществлялось 1200 вызовов, и это число значительно превышало ожидания. Следовательно, команда обновила коэффициент масштабирования до актуального значения.

Практический пример: фитнес-функция соответствия

Команда из предыдущего примера столкнулась с проблемой совместного использования паттерна Strangler Fig — как проверить, что новая система воспроизводит поведение старой? Они построили *фитнес-функцию соответствия* (fidelity fitness function), которая позволяла всем командам выборочно заменять фрагменты функциональности по одному за раз. Большинство таких фитнес-функций построены по образцу в примере 4.11 и обеспечивают возможность

запускать две версии кода одновременно (с пороговыми значениями), чтобы убедиться, что новая повторяет старую.

Фитнес-функция соответствия, реализованная командой, показана на рис. 9.10.

Разработчики внедрили фитнес-функцию для обеспечения согласованности. Однако в итоге они получили и побочное преимущество, выявив данные, поступавшие из незадокументированных источников и позволившие составить целостную картину зависимостей данных в (плохо документированной) унаследованной системе.

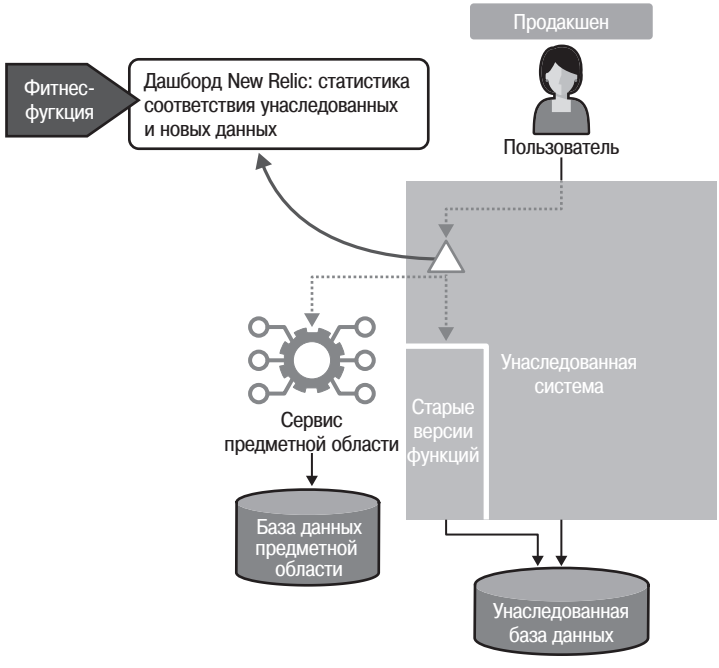


Рис. 9.10. Фитнес-функция соответствия для обеспечения эквивалентных ответов

Построение фитнес-функций предприятия

Роль архитектора эволюционной системы предприятия состоит в создании *управляющих фитнес-функций на уровне всего предприятия*. Подобную изменяемую модель можно реализовать в архитектуре микросервисов. Поскольку каждый сервис эксплуатируется независимо от других, в такой модели нет потребности в совместном использовании ресурсов. Вместо этого архитектору необходимо подобрать надлежащие точки сцепления в архитектуре (например, шаблоны сервисов) и подходящую платформу. Владельцем этой функции

совместно используемой инфраструктуры обычно является архитектура предприятия, и выбор платформ в ней ограничен теми, которые поддерживаются во всех частях предприятия.

Практический пример: уязвимость нулевого дня

Что делать, если в одном из используемых фреймворков или библиотек разработки обнаруживается уязвимость нулевого дня? Инструментов сканирования для поиска известных уязвимостей на уровне сетевых пакетов великое множество, но в них часто нет подходящих ловушек (хуков) для своевременного тестирования. Известен пример, когда подобное случилось в крупном финансовом учреждении несколько лет назад. 7 сентября 2017 года Equifax, ведущее американское бюро кредитных историй, объявило об утечке данных. Проблему проследили до взлома популярного веб-фреймворка Struts в экосистеме Java (Apache Struts vCVE-2017-5638). Его создатель опубликовал заявление, в котором сообщил об уязвимости, и выпустил патч 7 марта 2017 года. Отдел внутренней безопасности связался с Equifax и другими клиентами на следующий день, предупредив их о проблеме, и провел сканирование 15 марта 2017 года, которое выявило *большинство* затронутых систем... но не *все*. Таким образом, критический патч обновления не применялся ко многим старым системам вплоть до 29 июля 2017 года, когда эксперты по безопасности Equifax обнаружили взлом, который привел к утечке данных.

В среде с автоматизированным управлением для каждого проекта запускается пайплайн развертывания, в котором у специалистов по безопасности есть «слот» для развертывания фитнес-функций. В большинстве случаев они представляют собой обычные проверки безопасности, такие как проверка запрета хранить пароли в базах данных и тому подобные. Однако единый механизм позволяет добавить в каждый проект тестирование уязвимости нулевого дня, проверяющее соответствие фреймворка и номера версии; если оно выявляет скомпрометированную версию, сборка не выполняется и группа безопасности получает уведомление. Все чаще команды уделяют особое внимание цепочке поставок программ — происхождению библиотек и фреймворков, в особенности инструментов с открытым исходным кодом. К сожалению, очень часто именно инструменты разработчика становятся вектором атаки. Поэтому командам необходимо проверять метаданные зависимостей. К счастью, появился ряд средств для отслеживания и автоматизации управления цепочкой поставок программ, такие как snyk (https://snyk.io/) и Dependabot (https://github.com/dependabot), используемые GitHub.

Команды настраивают пайплайны развертывания на реагирование при любом изменении экосистемы: кода, схемы базы данных, конфигурации развертывания и фитнес-функций. Изменения зависимостей выступают как ловушка нужной

информации в нужное время, позволяя группе безопасности отслеживать возможные уязвимости.

Если пайплайн развертывания для применения фитнес-функций в процессе сборки используется в каждом проекте, архитекторы системы предприятия могут добавлять в него собственные фитнес-функции. Это обеспечивает постоянную проверку масштабируемости, безопасности и других параметров уровня предприятия и обнаружение проблем на самых ранних этапах. Подобно тому как проекты микросервисов используют шаблоны сервисов для унификации частей технической архитектуры, архитекторы систем предприятий могут использовать пайплайны развертывания для последовательного тестирования всего проекта.

Подобные механизмы позволяют повсеместно автоматизировать важные задачи управления предприятием и управлять критическими и важными аспектами программной разработки. Современное ПО состоит из огромного количества подвижных компонентов, что требует автоматизации для обеспечения его надежной работы.

Выделение ограниченных контекстов в существующей интеграционной архитектуре

В разделе «Паттерны повторного использования» на с. 143 (глава 5) мы обсуждали проблемы, возникающие при попытках избежать хрупкости архитектуры при повторном использовании. На практике такая проблема часто возникает при синхронизации повторного использования на уровне предприятия и изоляции с помощью ограниченных контекстов и квантов архитектуры, обычно на уровне данных, как показано на рис. 9.11.

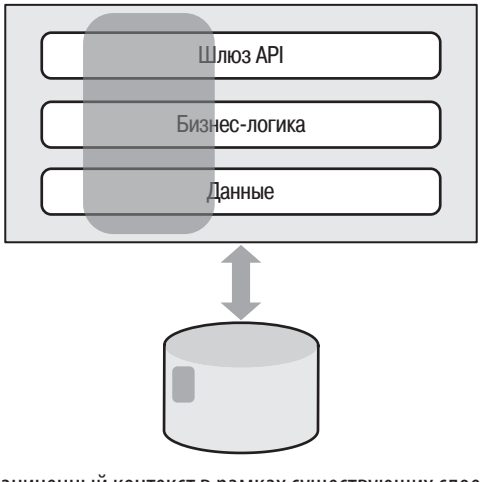


Рис. 9.11. Ограниченный контекст в рамках существующих слоев архитектуры

Многоуровневая архитектура — часто используемый паттерн, в котором компоненты выделяются по техническому принципу: представление, персистентность и т. д. В основе многоуровневой архитектуры лежит разделение ответственности, которое (надеемся) увеличивает возможности для повторного использования. *Техническое разделение* — это построение архитектуры по техническому принципу; долгое время это был самый распространенный стиль архитектуры.

С появлением DDD архитекторы начали активно использовать его принципы, особенно ограниченный контекст. Две самые распространенные современные топологии архитектурных решений — это *модульные монолиты* и *микросервисы*, и обе они в значительной степени основаны на DDD.

Однако эти две модели принципиально несовместимы — многоуровневая архитектура способствует разделению ответственности и облегчает повторное использование в различных контекстах, и это одно из ее главных преимуществ. Но, как мы уже демонстрировали, такому сквозному повторному использованию препятствуют и связующее свойство локальности, и принцип, лежащий в основе ограниченного контекста.

Как же преодолеть это противоречие? Поддерживая разделение ответственности и не допуская разрушительных побочных эффектов сквозного повторного использования. По сути, это еще одно доказательство того, что архитектура требует управления, что вновь приводит нас к необходимости создания фитнес-функций, дополняющих структуру.

Рассмотрим архитектуру, показанную на рис. 9.12.

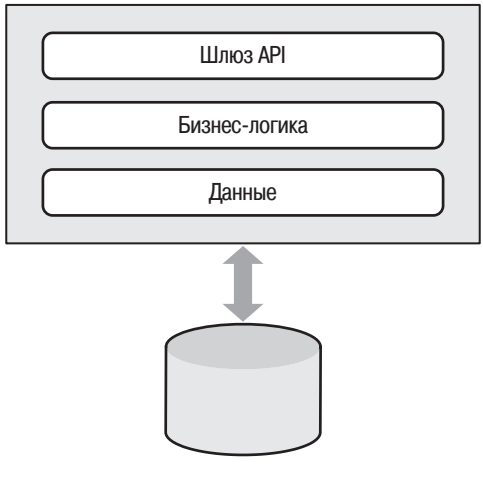


Рис. 9.12. Традиционная многоуровневая архитектура, включающая компоненты и монолитную базу данных

На рис. 9.12 архитектура разделена по техническому принципу, а не по фактическим уровням. Однако работая по принципам DDD, команда определила части приложения с интеграционной архитектурой, которые должны быть изолированы как ограниченный контекст, что показано на рис. 9.13.

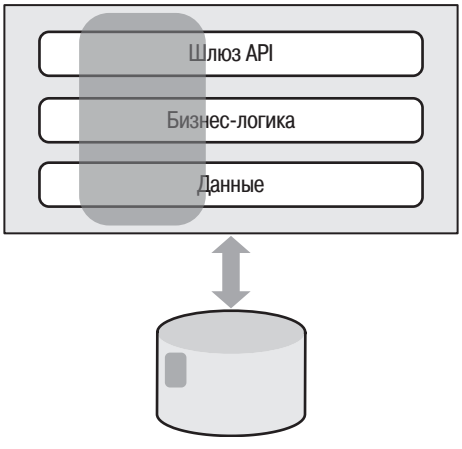


Рис. 9.13. Ограниченный контекст внутри другой архитектуры

На рис. 9.13 показан ограниченный контекст (область, выделенная темной заливкой) внутри технических уровней. Хотя дальнейшее разделение частей предметной области на основе технических характеристик не нарушит работу системы, необходимо также предотвратить сцепление приложений с деталями их реализации.

Следовательно, команда задает фитнес-функции, чтобы избежать взаимодействий, нарушающих границы контекста, как представлено на рис. 9.14.

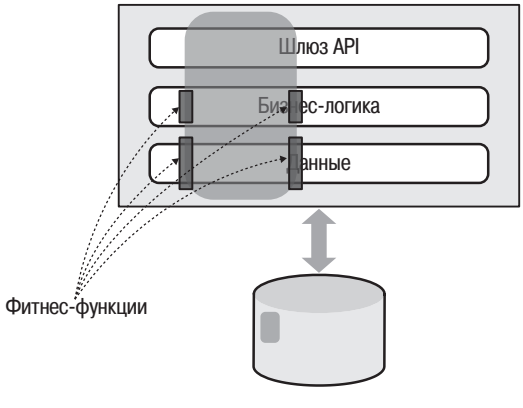


Рис. 9.14. Выделение ограниченного контекста в многоуровневой архитектуре

Фитнес-функции задаются везде, где необходимо предотвратить случайное сцепление. Конечно, возвращаясь к главной теме книги, точный вид этих фитнес-функций показать здесь невозможно, он будет определяться активами, которые необходимо защитить. Однако основная задача ясна: предотвратить нарушение ограниченного контекста из-за нарушения принципа локальности коннасценции.

С чего начать?

Многие архитекторы, пытающиеся преобразовать архитектуры, подобные большим комкам грязи, в эволюционные, не могут понять, с чего начинать эту работу. Хотя чаще всего в первую очередь необходимо обеспечить надлежащий уровень связанности и модульности, иногда первоочередную важность имеют и другие параметры. Например, если схема данных связана до такой степени, что ее невозможно разделить, прежде всего стоит определить, как администраторы БД могут добиться модульности. В этом разделе мы разберем некоторые популярные стратегии и мотивы внедрения эволюционных архитектур.

Низко висящие фрукты

Если компании нужна быстрая победа, чтобы доказать правильность выбранного эволюционного подхода, можно внедрить самую простую реализацию, которая подтвердит его эффективность. Как правило, это будет изменение части системы, которая уже в достаточной степени разделена и не имеет критического влияния на какие-либо зависимости. Увеличение модульности и уменьшение связанности позволит реализовать и другие возможности эволюционной архитектуры, а именно фитнес-функции и инкрементные изменения. Более высокая степень разделения позволяет проводить более целенаправленное тестирование и создавать фитнес-функции. Лучшая изоляция развертываемых единиц упрощает создание пайплайнов развертывания и более надежных тестов.

В инкрементно изменяемых средах метрики служат обычным дополнением к пайплайнам развертывания. При использовании для проверки концепции соответствующие метрики необходимо собирать для сценариев «до» и «после». Лучший способ проверить правильность подхода — собрать конкретные данные; помните, что *важно не обсуждать, а подтвердить*.

Подход «сначала самое простое» позволит минимизировать риски за счет возможного снижения ценности, если команде не удастся добиться и *простоты*, и *высокой ценности* одновременно. Эта стратегия подходит компаниям, которые настроены скептически и не готовы сразу полностью переходить к эволюционной архитектуре.

Сначала самое ценное

Альтернативой подходу «сначала самое простое» служит подход «сначала самое ценное»: найти критическую часть системы и сделать эволюционной именно ее. Эту стратегию можно использовать в нескольких случаях. Прежде всего, если архитекторы твердо уверены в необходимости создания эволюционной архитектуры, изменение наиболее ценной ее части в первую очередь послужит доказательством их приверженности этой идее. Если же компания еще не определилась с выбором в пользу эволюционности, архитекторы смогут оценить применимость этой идеи в их экосистеме. Изменяя наиболее ценную ее часть, они создадут условия для долгосрочной реализации ценности эволюционной архитектуры. И в третьем сценарии, если архитекторы сомневаются, что эти идеи будут работать в их приложении, проверяя концепции в наиболее ценной части системы, они получат объективные данные о том, стоит ли продолжать их внедрение.

Тестирование

Многие компании с сожалением признают, что проводят недостаточный объем тестирования систем. Если в кодовой базе мало тестов или они вообще отсутствуют, то, прежде чем начинать масштабный переход к эволюционной архитектуре, стоит добавить некоторые критические тесты.

К проектам, в рамках которых в кодовую базу добавляются только тесты, обычно относятся с подозрением, особенно если внедрение новых функций при этом откладывается. Лучше, если высокоуровневое функциональное тестирование сочетается с повышением модульности. Обертывание функциональности модульными тестами облегчает внедрение таких практик, как разработка на основе тестирования (TDD), но требует времени на реализацию. Рациональнее перед реструктуризацией кода добавить в него крупномодульные функциональные тесты поведения, что позволит проверить, что реструктуризация не повлияла на поведение системы в целом.

Тестирование — критически важное условие инкрементности эволюционной архитектуры, тесты активно используются для задания фитнес-функций. Поэтому, чтобы реализовать в архитектуре возможности фитнес-функций и инкрементности, необходимо обеспечить в ней по крайней мере базовый уровень тестирования, и простота реализации эволюционной архитектуры напрямую зависит от того, насколько полно архитектура покрывается тестами.

Инфраструктура

Не все компании открыты новым возможностям, и чаще всего отсутствие инноваций влияет на работу группы эксплуатации. В компаниях с неработающей инфраструктурой решение этой проблемы может способствовать переходу

к эволюционной архитектуре. Проблемы инфраструктуры имеют разные формы. Например, часть компаний передает эксплуатацию на аутсорсинг и перестает контролировать этот критически важный элемент экосистемы; когда возникают накладные расходы на координацию работы разных компаний, сложность DevOps возрастает на порядки.

Еще одна распространенная инфраструктурная проблема — отсутствие прозрачности между группами разработки и эксплуатации, когда разработчики не имеют представления, как на практике работает их код. Такая ситуация характерна для забюрократизированных компаний, в которых каждый отдел действует автономно.

ИНФРАСТРУКТУРА ВСЕГДА ВЛИЯЕТ НА АРХИТЕКТУРУ

Однажды Нил консультировал компанию, предоставлявшую услуги хостинга. У компании было большое количество серверов (около 2500 на тот момент), и *внутри* группы эксплуатации существовали независимые подразделения: одна команда устанавливала оборудование, другая — операционные системы, а третья — приложения. Излишне говорить, что когда разработчику требовался ресурс, он отправлял заявку в черную дыру группы эксплуатации, где эти заявки накапливались и передавались по кругу неделями, пока не появлялись ресурсы. Проблема усугублялась тем, что за год до этого ИТ-директор ушел в отставку, и его функции выполнял финансовый директор. Конечно, в первую очередь его заботила экономия средств, а не модернизация того, что он считал накладными расходами.

Изучая слабые места в работе, один из разработчиков заметил, что на каждом сервере размещается не более пяти пользователей, что было просто чудовищно, учитывая простоту приложения. Разработчики стыдливо признались, что чрезвычайно злоупотребляли состоянием сеанса HTTP, по сути сделав из него огромную базу данных в памяти. Это привело к тому, что на одном сервере размещалось всего несколько пользователей. Проблема заключалась в том, что группа эксплуатации не могла создать реалистичную рабочую среду для отладки и из-за политики компании категорически запретила разработчикам проводить отладку (или хотя бы обширный мониторинг) на продакшен. Не имея реалистичной версии приложения, разработчики были не в силах распутать постепенно растущий клубок.

Проведя кое-какие расчеты, мы выяснили, что компании было бы достаточно иметь на порядок меньше серверов, примерно 250. Однако она была слишком занята покупкой новых серверов, установкой операционных систем и т. д. По иронии судьбы, экономия влетела ей в копеечку.

В конце концов у разработчиков не осталось иного выхода, кроме как создать свою собственную партизанскую группу DevOps и самостоятельно управлять серверами, полностью в обход традиционной модели организации. Это грозило обернуться противоречиями между двумя группами, но в краткосрочной перспективе помогло начать постепенную реструктуризацию приложения.

Наконец, в некоторых организациях архитекторы и разработчики игнорируют передовой опыт и, как следствие, накапливают огромные объемы технического долга, который проявляется в инфраструктуре. Существуют компании, даже не имеющие представления о том, что где работает, как и других базовых знаний о взаимодействиях между архитектурой и инфраструктурой.

Все данные здесь советы в конечном счете сводятся к одному условию: *все зависит от обстоятельств*! Только архитекторы, разработчики, администраторы БД, DevOps, тестировщики, специалисты по безопасности и другие участники процесса могут предложить лучшую возможную дорожную карту эволюционной архитектуры.

Практический пример: архитектура системы предприятия в PenultimateWidgets

Компания PenultimateWidgets изучает возможность модернизации значительной части своей платформы, и команда архитекторов системы предприятия составила электронную таблицу свойств, которыми должна обладать новая платформа: безопасность, показатели производительности, масштабируемость, возможность развертывания и многие другие. Каждая категория включала от 5 до 20 определенных критериев. Например, одна из метрик времени работы требовала, чтобы доступность каждого сервиса поддерживалась на уровне пяти девяток (99,999). В общей сложности архитекторы определили 62 требуемых свойства.

Но они также поняли, что такое количество чревато проблемами. Во-первых, как на проектах проверять каждое из этих 62 свойств? Можно внедрить политику, но кто будет заниматься проверкой ее соответствия, да еще постоянно? Проверять это вручную, даже отдельному сотруднику, будет очень сложно.

Во-вторых, имеет ли смысл вводить строгие требования к доступности в каждой части системы? Так ли важно, чтобы системный администратор всегда видел эти пять девяток? Внедрение общих политик часто оборачивается чрезмерным техническим усложнением системы.

Чтобы решить эти проблемы, архитекторы выразили критерии в виде фитнес-функций и создали шаблон пайплайна развертывания для каждого проекта. Внутри пайплайна развертывания архитекторы разработали фитнес-функции для автоматической проверки критически важных характеристик, таких как безопасность, оставив командам возможность добавлять специфические фитнес-функции (например, доступность) для своих сервисов.

Что дальше?

Каким будет будущее эволюционной архитектуры? Знакомясь с ее принципами и практиками, разработчики будут внедрять их в текущие операции и использовать для создания новых возможностей, таких как разработка на основе данных.

Нас ждет еще много работы по созданию более сложных фитнес-функций, но ряд компаний уже успешно применяют имеющиеся для решения текущих задач и обеспечения свободного доступа к своим решениям. В начале эпохи agile люди жаловались на то, что определенные проблемы слишком сложно автоматизировать, но разработчики бесстрашно шли вперед, и теперь автоматизировать можно целые центры обработки данных. В частности, компания Netflix добилась огромных успехов в разработке концепции и создании таких инструментов, как Simian Army, который поддерживает целостные непрерывные фитнес-функции (хотя компания так их еще не называла).

Перспективных направлений несколько.

Фитнес-функции на основе искусственного интеллекта

Для постоянно поддерживаемых проектов возникают и развиваются крупные фреймворки на основе искусственного интеллекта с открытым исходным кодом. По мере того как эти инструменты войдут в обиход разработчиков ПО, архитекторы предвидят появление фитнес-функций на основе ИИ, выявляющих аномальное поведение. Компании, занимающиеся выпуском и обслуживанием кредитных карт, уже применяют эвристику, например для пометки почти одновременных транзакций в разных частях света; архитекторы могут начать создавать инструменты для поиска нестандартного поведения в архитектуре.

Генеративное тестирование

Генеративное тестирование — практика, применяемая во многих сообществах функционального программирования и набирающая популярность. В традиционном модульном тестировании каждый тест включает предположение о том, какой результат считать верным. Однако практика генеративного тестирования подразумевает запуск большого количества тестов, фиксирование результатов и их дальнейший статистический анализ для поиска отклонений. Рассмотрим, к примеру, обычную проверку границ числовых диапазонов. Традиционные модульные тесты проверяют известные места возможных сбоев (отрицательные числа, целочисленное переполнение и т. д.), но могут не учесть непредвиденные граничные случаи. Генеративные тесты проверяют все возможные значения и выявляют граничные случаи, в которых происходит сбой.

Почему (или почему нет)

Серебряной пули нет, в том числе в архитектуре. Необязательно делать каждый проект эволюционным, принимая на себя дополнительные расходы и усилия, если это не принесет ему пользу.

Зачем создавать эволюционную архитектуру

Многие компании отмечают, что цикл изменений ускорился за последние несколько лет, что отражает приведенное выше замечание *Forbes* о том, что каждая компания должна уметь разрабатывать и поставлять программное обеспечение. Рассмотрим, почему все же стоит создавать эволюционную архитектуру.

Предсказуемость или эволюционность

Для многих компаний важно долгосрочное планирование ресурсов и других стратегических вопросов; им важна *предсказуемость*. Однако из-за динамического равновесия экосистемы программной разработки предсказуемость утратила свою актуальность. Планы, которые продолжают строить архитекторы систем предприятий, могут быть нарушены в любой момент.

Даже компаниям из стабильных отраслей не стоит игнорировать опасности, которым подвергаются системы, неспособные эволюционировать. Сложившаяся международная индустрия такси с вековой историей в итоге испытала потрясение, когда на рынок вышли компании, предоставляющие услуги каршеринга, которые были готовы к изменениям экосистемы и своевременно отреагировали на них. Явление, известное как «дилемма инноватора», пророчит неудачу компаниям на устоявшихся рынках, поскольку более гибкие стартапы обычно лучше себя чувствуют в меняющейся экосистеме.

Создание эволюционной архитектуры требует времени и усилий, но результатом этих вложений становится способность компании реагировать на существенные изменения на рынке без серьезной перестройки своих внутренних процессов. Эпоха предсказуемых мейнфреймов и специализированных операционных центров осталась позади. Крайне изменчивый характер мира разработки стимулирует организации к переходу на инкрементные изменения.

Масштабируемость

В течение некоторого времени лучшей практикой в архитектуре было создание транзакционных систем на основе реляционных баз данных, координацию в которых обеспечивали разнообразные функции БД. Недостаток такого под-

хода заключается в трудности масштабирования базы данных бэкенда. Для решения этой проблемы был разработан ряд изощренных технологий, но они лишь снимали симптомы, а фундаментальная проблема масштабирования — связанность — оставалась актуальной. Любые точки связанности в архитектуре затрудняют масштабирование, и координация на уровне БД в конечном итоге заводит разработчиков в тупик.

С этой проблемой столкнулась и компания Amazon. Ее официальный сайт представлял собой монолитный фронтенд, связанный с монолитным бэкендом на основе баз данных. Когда трафик увеличился, разработчикам пришлось масштабировать базы данных. В какой-то момент они достигли предела возможностей масштабирования, и это сказалось на производительности сайта — страницы стали загружаться медленнее.

Amazon осознала, что привязка всех компонентов к *чему-то одному* (к реляционной базе данных, корпоративной сервисной шине и т. д.) ограничивает масштабируемость. Перепроектировав архитектуру в стиле микросервисов, чтобы устранить лишнюю связанность, Amazon вернула возможность масштабирования экосистемы.

Подобное разделение обладает дополнительным преимуществом — оно повышает способность архитектуры к эволюции. На протяжении всей книги мы подчеркивали, что самым большим препятствием для обеспечения эволюционности является избыточная связанность. Обеспечение масштабируемости системы, как правило, способствует ее эволюционности.

Расширенные возможности для бизнеса

Многие организации завистливо посматривают на Facebook, Netflix и другие передовые технологические компании, поскольку они способны внедрять сложные функции. Инкрементные изменения позволяют использовать такие популярные практики, как гипотезы и разработка на основе данных. Многие компании стараются включить своих пользователей в цикл обратной связи с помощью многомерного тестирования. Архитектура, способная развиваться, выступает важнейшим условием реализации многих передовых практик DevOps. Например, разработчикам сложно проводить A/B-тестирование, если между компонентами существует сильная связанность, поскольку это затрудняет разделение ответственности. В целом эволюционная архитектура способна эффективнее реагировать на неизбежные, но непредсказуемые изменения.

Время цикла как бизнес-метрика

В разделе «Пайплайн развертывания» на с. 57 (глава 3) мы установили различие между *непрерывной доставкой*, когда хотя бы один этап в пайплайне развертывания выполняется вручную, и *непрерывным развертыванием*, когда каждый этап в случае успешного завершения автоматически переходит на следующий. Внедрить непрерывное развертывание довольно сложно — зачем компании прилагать такие усилия?

Дело в том, что на отдельных рынках определяющим фактором бизнеса стало время цикла. Некоторые крупные консервативные компании относятся к разработке как к накладным расходам и стараются минимизировать затраты. Инновационные компании, напротив, видят в ней возможности для конкурентного преимущества. Например, если AcmeWidgets создала архитектуру, в которой время цикла составляет три часа, а цикл PenultimateWidgets по-прежнему занимает шесть недель, AcmeWidgets получает преимущество, которое она может использовать.

Многие компании сделали время цикла одной из важнейших бизнес-метрик, поскольку на их рынке высокая конкуренция. Все рынки в конечном итоге становятся конкурентными. Например, в начале 1990-х годов некоторые крупные компании настойчивее остальных стремились автоматизировать ручные процессы с помощью программных средств и получили огромное преимущество, поскольку в конечном итоге эту необходимость осознал весь бизнес.

Изолирование характеристик архитектуры на уровне квантов

Преобразование традиционных нефункциональных требований к виду фитнес-функций и создание хорошо инкапсулированных квантов архитектуры позволит поддерживать различные характеристики архитектуры в каждом кванте, что является одним из преимуществ микросервисов. Поскольку техническая архитектура каждого кванта отделена от других квантов, для разных случаев использования можно подбирать разные архитектуры. Например, разработчики небольшого сервиса могут выбрать микроядерную архитектуру, потому что им требуется поддерживать небольшое ядро и постепенно его увеличивать. Другая команда может выбрать событийно-управляемую архитектуру из-за требования к масштабируемости. Если бы оба сервиса были частью монолита, архитекторам пришлось бы идти на компромиссы, чтобы удовлетворить оба требования. Изолируя техническую архитектуру на уровне небольшого кванта, архитекторы могут работать с основными характеристиками отдельного кванта, без необходимости анализировать компромиссы для реализации противоречащих друг другу приоритетных требований.

Адаптация или эволюционность

Многие организации со временем оказываются в ловушке постепенно увеличивающегося технического долга и нежелания проводить необходимую реструктуризацию, что ведет к хрупкости систем и точек интеграции. Чтобы справиться с этой хрупкостью, компании применяют такие инструменты подключения, как сервисные шины, что частично облегчает головную боль, но не способствует более глубокой логической связности (cohesion) бизнес-процессов. Использование сервисной шины — это пример *адаптации* существующей системы к изменившимся условиям. Но, как мы уже отмечали, побочным эффектом адаптации является увеличение технического долга. Когда разработчики что-то адаптируют, они сохраняют имеющееся поведение и добавляют к нему новое. Чем больше циклов адаптации переживает компонент, тем больше параллельных моделей поведения в нем появляется, что ведет к увеличению сложности (будем надеяться, это делается рационально).

Использование переключателей функций — хороший пример пользы адаптации. Часто разработчики используют переключатели при проверке вариантов в рамках разработки, основанной на гипотезах, проводя тестирование с привлечением пользователей, чтобы выяснить, что вызывает у них наибольший отклик. В этом случае технический долг, вызванный переключениями, желателен и создается целенаправленно. Конечно, такие переключатели следует удалять после принятия решения.

Напротив, *эволюция* подразумевает фундаментальные изменения. Эволюционная архитектура — это архитектура, изменяемая по месту и защищаемая от сбоев с помощью фитнес-функций. Ее желаемая реализация — система, которая постоянно и продуктивно эволюционирует, не накапливая в себе скрытые устаревшие решения.

В каких случаях эволюционная архитектура не нужна?

Эволюционная архитектура — это не панацея! Иногда у компаний есть убедительные основания отказаться от этой идеи. Рассмотрим самые частые из них.

Большие комки грязи, неспособные к эволюции

Одной из основных «-остей», которыми пренебрегают архитекторы, является *выполнимость* — стоит ли вообще браться за проект? Если архитектура представляет собой безнадежно связанный большой комок грязи, то обеспечение ее эволюционности потребует огромного объема работы — скорее всего, будет проще переписать архитектуру заново. Компании с неохотой отказываются от

вещей, представляющих хоть какую-то видимую ценность, но часто их переделка обходится дороже, чем переписывание.

Как определить, что компания попала именно в такую ситуацию? Преобразование существующей архитектуры в эволюционную начинается с рассмотрения *модульности*. Таким образом, первая задача разработчика — найти в имеющейся системе все модульные элементы и перестроить архитектуру на их основе. Это поможет несколько распутать архитектуру, и тогда архитекторы смогут изучить ее базовые структуры и сделать выводы о целесообразности ее реструктуризации.

Преобладание других характеристик архитектуры

Эволюционность — лишь одно из условий, которые архитекторы должны учитывать при выборе архитектурного стиля. Ни одна архитектура не способна в полной мере поддерживать противоречащие друг другу цели. Например, в одной архитектуре трудно совместить высокую производительность и высокую масштабируемость. Иногда для архитектуры важнее поддерживать другие свойства, нежели способность к эволюции.

Чаще всего архитекторы выбирают архитектуру для поддержки широкого набора требований, например высокой доступности, безопасности и масштабирования. Этим обуславливается выбор популярных архитектурных моделей, таких как монолит, микросервисы или модели на основе событий. Однако существуют и *предметно-ориентированные архитектуры,* построенные вокруг конкретной характеристики. Сделать эволюционной такую архитектуру, созданную для одной цели, будет очень сложно (если только разработчикам не повезет и их задачи не совпадут с задачами архитектуры). Таким образом, большинству предметно-ориентированных архитектур не требуется эволюционность, поскольку в первую очередь они призваны удовлетворять поставленной цели.

Жертвенная архитектура

Мартин Фаулер определил жертвенную архитектуру как архитектуру, которую создают, чтобы впоследствии удалить. Многие компании на начальном этапе разработки создают простые версии продукта для изучения рынка или проверки жизнеспособности продукта. Если продукт такую проверку пройдет, компании переходят к построению *настоящей* архитектуры, поддерживающей выявленные важные характеристики.

Такой подход применяется в стратегических целях. Жертвенные архитектуры характерны для минимально жизнеспособного продукта, создаваемого в целях тестирования рынка, чтобы затем в случае успеха построить более надежную

архитектуру. Жертвенность подразумевает, что такую архитектуру не будут развивать, а заменят постоянной, когда наступит такая необходимость. Развитие облачных технологий повышает привлекательность этого варианта для компаний, тестирующих новые рынки или предложения.

Скорое закрытие бизнеса

Эволюционная архитектура помогает компаниям адаптироваться к изменениям экосистемы. Если компания не планирует продолжать работу уже через год, нет смысла закладывать в ее архитектуру способность к эволюции.

Итоги

Создать эволюционную архитектуру невозможно, просто загрузив и запустив некий универсальный набор инструментов. Для этого потребуется применить целостный подход к управлению архитектурой на основе инженерного опыта. В основе настоящей программной инженерии лежат автоматизация и инкрементные изменения — характеристики эволюционной архитектуры.

Помните, что чаще всего готовых инструментов для экосистемы не существует. Поэтому главное для вас — ответить на вопрос: есть ли где-то информация, которая мне нужна? Если она есть, то, чтобы она стала представлять собой архитектурную ценность, достаточно собрать и агрегировать ее с помощью простого ручного скрипта.

Фитнес-функции необязательно должны быть сложными. Как и в модульном тестировании предметной области, фитнес-функции могут проверять важнейшие характеристики архитектуры, чтобы оправдать усилия по их созданию и поддержке. Нельзя говорить о конечном состоянии эволюции в архитектуре — можно только о том, какую ценность добавляет в нее эволюционный подход.

Чтобы разработать программную систему, архитекторы должны быть уверены, что ее структура соответствует применяемым инженерным практикам. Контроль связанности и автоматизация проверок — ключ к созданию хорошо управляемых архитектур, которые способны эволюционировать в предметной области, в техническом аспекте или в двух этих измерениях одновременно.

Об авторах

Нил Форд — директор, архитектор ПО и идейный вдохновитель в ThoughtWorks, компании-разработчике программного обеспечения, объединяющей увлеченных, целеустремленных людей с нестандартным мышлением, что помогает создавать технологии для решения самых сложных задач, производящие революцию в ИТ-индустрии и стимулирующие позитивные перемены в обществе. До прихода в ThoughtWorks Нил был директором по технологиям в DSW Group, Ltd., компании национального уровня, специализирующейся на тренингах и развитии.

Нил имеет степень в области computer science Университета штата Джорджия (со специализацией на языках и компиляторах) и дополнительную степень в области математики (со специализацией на статистическом анализе). Он заслужил международное признание как эксперт в области разработки и доставки программного обеспечения, в частности на стыке agile-методологии и программной архитектуры. Нил является автором ряда статей в тематических журналах, девяти книг (и их число продолжает расти) и десятков видеопрезентаций, а также спикером сотен конференций разработчиков по всему миру. Его работы посвящены вопросам программной архитектуры, непрерывной доставки, функционального программирования и инноваций в сфере ПО, а кроме того, у него есть книга, ориентированная на бизнес, и видео на тему улучшения качества технических презентаций. Он является консультантом в сфере проектирования и создания крупномасштабных корпоративных приложений. Если вы хотите узнать больше о сферах интересов Нила, посетите его веб-сайт nealford.com.

Доктор Ребекка Парсонс — технический директор ThoughtWorks, имеющая многолетний опыт разработки приложений для различных отраслей и систем. Ее технические компетенции включают руководство созданием крупномасштабных распределенных объектных приложений, интеграцию разрозненных систем и работу с командами архитекторов. Помимо своей глубокой страсти к технологиям она активно продвигает разнообразие в технологической отрасли.

До прихода в ThoughtWorks д-р Парсонс являлась доцентом компьютерных наук в Университете Центральной Флориды, где вела курсы по компиляторам, оптимизации программ, распределенным вычислениям, языкам программиро-

вания, теории вычислений, машинному обучению и вычислительной биологии. В качестве постдокторанта директора Лос-Аламосской национальной лаборатории д-р Парсонс занималась исследованиями в сферах параллельных и распределенных вычислений, генетических алгоритмов, вычислительной биологии и нелинейных динамических систем.

Ребекка Парсонс получила степень бакалавра компьютерных наук и экономики в Университете Брэдли, степень магистра компьютерных наук в Университете Райса и степень PhD в области компьютерных наук в Университете Райса. Она также является соавтором книг «Domain-Specific Languages, The ThoughtWorks Anthology» и первого издания книги «Эволюционная архитектура».

Патрик Куа — независимый тренер технических директоров, бывший технический директор Компании N 26 и бывший главный технический консультант ThoughtWorks, проработавший в технологической отрасли более 20 лет. Свою личную миссию он видит в том, чтобы способствовать росту технических лидеров с помощью индивидуального коучинга, онлайн-семинаров и очных семинаров по техническому лидерству, а также своего популярного информационного бюллетеня *Level Up* для лидеров в области технологий.

Он является автором книг «The Retrospective Handbook: A Guide for Agile Teams» и «Talking with Tech Leads: From Novices to Practitioners» и предлагает обучение в Академии техлидов *The Tech Lead Academy*.

Вы можете узнать о нем больше на его веб-сайте patkua.com или связаться с ним в твиттере @patkua.

Прамод Садаладж — директор отдела данных и DevOps в ThoughtWorks, где решает нестандартную задачу преодоления разрыва между профессионалами в области баз данных и разработчиками приложений. Чаще всего он работает с клиентами, чьи запросы в отношении данных особенно сложны и требуют применения новых технологий и методов. В начале 2000-х создал методологию эволюционной разработки реляционных баз данных на основе миграции схем с контролем версий.

Является соавтором книг «Software Architecture: The Hard Parts»[1], «Refactoring Databases», «NoSQL Distilled», а также автором «Recipes for Continuous Database Integration» и продолжает освещать новые идеи, которые приходят к нему и его клиентам.

[1] *Форд Н.* и др. «Современный подход к программной архитектуре: сложные компромиссы». СПб., издательство «Питер».

Иллюстрация на обложке

Животное, изображенное на обложке, — открытый мозговой коралл *(Trachyphyllia geoffroyi)*. Этот коралл с крупными полипами, также известный как складчатый или кратерный, обитает в Индийском океане.

Такие свободноживущие кораллы, выделяющиеся своими характерными складками, яркой окраской и выносливостью, днем питаются за счет продуктов фотосинтеза произрастающих на них зооксантелловых водорослей, а ночью расправляют свои ловчие щупальца в поисках добычи — различных видов планктона и мелкой рыбешки — и отправляют ее в один из своих ртов (у некоторых подобных кораллов два или три рта).

Благодаря своему нарядному виду и простоте содержания *Trachyphyllia geoffroyi* часто выбирают для аквариумов, где они разрастаются в нижнем слое песка и/или ила, напоминающем мелководье морского дна в их естественной среде обитания. Комфортнее всего они себя чувствуют в среде с умеренным течением воды, богатой растительной и животной пищей.

Вид *Trachyphyllia geoffroyi* занесен в Красный список МСОП как находящийся под угрозой исчезновения. Многие животные с обложек изданий O'Reilly имеют подобный статус; все они важны для нашего мира.

Иллюстрацию для обложки выполнила Карен Монтгомери (Karen Montgomery) на основе старинной линейной гравюры из книги Жана Венсана Феликса Ламуру (Jean Vincent Félix Lamouroux) «Exposition méthodique des genres de l'ordre des Polypiers».