

O'REILLY®

ЭВОЛЮЦИОННАЯ АРХИТЕКТУРА

ПОДДЕРЖКА НЕПРЕРЫВНЫХ ИЗМЕНЕНИЙ



Нил Форд, Ребекка Парсонс, Патрик Куа

 ПИТЕР®

Building Evolutionary Architectures

Support Constant Change

Neal Ford, Rebecca Parsons, and Patrick Kua

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Нил Форд, Ребекка Парсонс, Патрик Куа

ЭВОЛЮЦИОННАЯ АРХИТЕКТУРА

ПОДДЕРЖКА НЕПРЕРЫВНЫХ ИЗМЕНЕНИЙ



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

2019

ББК 32.973.2-018
УДК 004.45
Ф79

Форд Нил, Парсонс Ребекка, Куа Патрик

Ф79 Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2019. — 272 с.: ил. — (Серия «Бестселлеры O'Reilly»).
ISBN 978-5-4461-0995-1

Пора по-новому взглянуть на постулаты, остававшиеся неизменными на протяжении многих лет. Динамично меняющийся мир диктует свои правила, в том числе и в компьютерной архитектуре. Происходящие изменения требуют новых подходов, заставляют жесткие системы становиться гибкими и подстраиваться под новые условия. Возможно ли долгосрочное планирование, если все непрерывно меняется? Как предотвратить постепенное ухудшение архитектурного решения с течением времени? Здесь вы найдете ответы и рекомендации, которые позволят защитить самые важные характеристики проекта в условиях непрерывных изменений.

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

ББК 32.973.2-018
УДК 004.45

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491986363 англ.

Authorized Russian translation of the English edition Building Evolutionary Architectures © 2017 Neal Ford, Rebecca Parsons, and Patrick Kua This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

ISBN 978-5-4461-0995-1

© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Бестселлеры O'Reilly», 2019
© Перевод с английского языка, Демьяников А. И., 2018

Оглавление

Предисловие	10
Введение	13
Типографские соглашения	13
От научного редактора перевода	14
Как связаться с нами	14
Дополнительная информация.....	15
От издательства.....	15
Благодарности.....	15
Глава 1. Архитектура программного обеспечения	17
Архитектура с эволюционным развитием	20
Как можно осуществлять долгосрочное планирование, если все постоянно меняется?.....	20
Как можно защитить созданную архитектуру от постепенной деградации?.....	24
Инкрементные изменения.....	26
Управляемое изменение	27
Многочисленные области архитектуры	28
Закон Конвея.....	33
Почему эволюционное развитие?	37
Краткие выводы	38
Глава 2. Функции пригодности.....	39
Что собой представляет функция пригодности?.....	42
Категории	45
Атомарная и комплексная функции	45
Триггерные и непрерывные функции.....	46

Статические и динамические функции	47
Автоматизированная и ручная функции	48
Временная функция	49
Функция с преднамеренным развитием	50
Предметно-ориентированная функция	50
Ранняя идентификация функций пригодности.....	50
Пересмотр функций пригодности	53
Глава 3. Проектирование инкрементных изменений	55
Строительные блоки.....	59
Тестопригодность	62
Конвейеры развертывания	64
Комбинирование категорий функций пригодности	70
Практический пример: реструктуризация архитектуры при ее развертывании 60 раз в день	73
Конфликтующие цели	76
Практический пример: добавление функций пригодности в сервис выставления счетов PenultimateWidgets	77
Разработка, основанная на гипотезах и на данных	81
Практический пример: что портировать?	84
Глава 4. Архитектурная связанность	86
Модульность.....	86
Квант и гранулярность архитектуры	87
Эволюция архитектурных стилей.....	92
Большой комок грязи.....	93
Монолитная архитектура	95
Событийно-ориентированная архитектура	106
Сервис-ориентированные архитектуры	113
Бессерверная архитектура	131
Контроль размера кванта	134
Практический пример: предотвращение циклов компонентов.....	135
Глава 5. Эволюционирующие данные	138
Эволюционное проектирование баз данных	139
Эволюционные схемы	139

Интеграция базы данных общего использования	142
Ненадлежащая связанность данных	148
Двухфазная фиксация транзакций	149
Возраст и качество данных.....	152
Практический пример: эволюционирование методов маршрутизации в PenultimateWidgets	154
Глава 6. Построение архитектуры с эволюционным развитием	157
Техники.....	158
1. Определить области, затрагиваемые эволюционным развитием	158
2. Определить для каждой области функцию(-и) пригодности	158
3. Использовать конвейер развертывания для автоматизации функций пригодности	159
Проекты с нуля	160
Настройка существующих архитектур	160
Надлежащие связанность и сцепление	160
Практики проектирования.....	161
Функции пригодности.....	162
Применение коммерческой продукции	164
Миграция архитектур	165
Шаги миграции	167
Эволюция модульных взаимодействий.....	171
Инструкции для построения эволюционирующей архитектуры	175
Удаление ненужной изменчивости	176
Сделайте решения обратимыми	179
Предпочтение следует отдавать эволюционированию, а не предсказуемости.....	180
Построение уровня защиты от повреждений	181
Практический пример: шаблоны сервисов.....	185
Построение жертвенной архитектуры	187
Уменьшить внешние изменения.....	189
Обновление библиотек и фреймворков	192
Отдавайте предпочтение непрерывной поставке, а не снимкам состояния системы	193
Версии внутренних сервисов	195

Практический пример: эволюционирование рейтингов PenultimateWidgets	196
------------------------------------------------------------------------------	-----

Глава 7. Архитектура с эволюционным развитием: ловушки и антипаттерны 200

Техническая архитектура	200
Антипаттерн: Vendor King	201
Ловушка: дырявая абстракция.....	203
Антипаттерн: ловушка на последних 10 %	206
Антипаттерн: неправильное повторное использование кода.....	208
Практический пример: принцип повторного использования в PenultimateWidgets.....	211
Ловушка: разработки ради резюме.....	213
Инкрементные изменения.....	213
Антипаттерн: ненадлежащее управление.....	214
Практический пример: модель управления «золотой середины» в PenultimateWidgets	217
Ловушка: недостаточная скорость для релиза	218
Проблемы бизнеса	221
Ловушка: адаптация продукта	221
Антипаттерн: составление отчетов.....	222
Ловушка: горизонты планирования.....	225

Глава 8. Внедрение эволюционной архитектуры 227

Организационные факторы	227
Кросс-функциональные команды.....	227
Организованные бизнес-возможности	230
Продукт важнее, чем проект	231
Работа с внешним изменением	234
Связи между участниками команды.....	235
Характеристики связей между командами.....	237
Культура.....	237
Культура эксперимента.....	239
Операционный денежный поток (OCF) и бюджетирование	242
Разработка функций пригодности для предприятия.....	244

Практический пример: PenultimateWidgets как платформа	246
С чего мы начнем?	246
Низко висящие фрукты	247
Максимальная ценность	247
Тестирование	248
Инфраструктура	249
Практический пример: архитектура предприятия в компании PenultimateWidgets	251
Будущее состояние?	252
Функции пригодности, использующие искусственный интеллект	252
Генеративное тестирование	253
Зачем это (или почему бы и нет)?	253
Зачем та или иная компания решает строить эволюционирующую архитектуру?	253
Практический пример: избирательный масштаб в PenultimateWidgets	257
По какой причине компания делает выбор не строить эволюционирующую архитектуру?	259
Убеждая других	262
Практический пример: консультация по системе дзюдо	262
Пример из бизнеса	263
«Будущее уже наступило...»	263
Двигаться быстро и без аварий	264
Меньше риска	264
Новые возможности	265
Построение архитектуры с эволюционным развитием	265
Об авторах	266
Выходные данные	269

Предисловие

Долгое время разработчики программного обеспечения придерживались мнения, что архитектуру программного обеспечения следует разрабатывать до написания первой строки кода. Под влиянием строительной отрасли считалось, что признаком успешной архитектуры программного обеспечения было то, что в ней не надо ничего менять в процессе разработки, это часто было связано с высокими затратами на переделку архитектуры.

В дальнейшем с появлением методов гибкого программного обеспечения такое представление об архитектуре изменилось. Метод заранее спланированной архитектуры основывался на том, что установленные требования не должны были меняться до написания кода, что приводило к поэтапному (или каскадному) методу проектирования, в котором за установленными требованиями следовала разработка архитектуры, а за ней, в свою очередь, следовала разработка программного обеспечения. Однако динамично меняющийся мир поставил под сомнение саму идею неизменности требований, так как изменения требований оказались просто необходимы для ведения бизнеса в современном мире и обеспечивали планирование проекта, позволяющее вносить контролируемые изменения.

В этом новом гибком мире многие люди стали задаваться вопросом о роли архитектуры. И конечно же, заранее планируемая архитектура не могла приспособиться к изменчивости современного мира. При создании архитектуры используется другой метод, который

гибким образом вносит необходимые изменения. При таком подходе разработка архитектуры выполняется в тесной связи с созданием программного обеспечения, так что архитектура может не только реагировать на изменение требований, но также принимать во внимание обратную связь от программирования. Мы обратимся к такой архитектуре с эволюционным развитием, чтобы показать, что даже в случае непредсказуемых изменений эта архитектура все еще может продолжать двигаться в правильном направлении.

Работая в компании *ThoughtWorks*, мы глубоко окунулись в это архитектурное мировоззрение. Ребекка вела многие из наших наиболее важных проектов в 2000-х и постепенно обеспечивала лидирующее положение, будучи руководителем технического отдела. Нил играл роль внимательного наблюдателя наших работ, синтезируя и передавая полученный нами опыт. Патрик совмещал свою работу над проектом с обеспечением нашего лидирующего положения в области техники. Мы всегда чувствовали жизненную важность создаваемой нами архитектуры и не могли довериться случаю. Совершая ошибки, мы получали бесценный опыт и начинали лучше понимать, как строить основу кода, которая могла бы эффективно реагировать на многие изменения своей цели.

В основе создания архитектуры с эволюционным развитием лежат небольшие изменения, которые с помощью обратной связи позволяют каждому узнать, как развивается система. Появление системы непрерывной поставки стало решающим фактором практической реализации архитектуры с эволюционным развитием. Авторское трио Нила, Ребекки и Патрика использует идею функций пригодности для управления состоянием архитектуры. Они изучают различные стили эволюционируемости архитектуры и делают акцент на проблемах, связанных с данными длительного хранения, — обычно этим пренебрегают. В большинстве пояснений, приведенных в этой книге, используется закон Конвея.

Несмотря на то что мы еще должны многое узнать о разработке архитектуры программного обеспечения эволюционирующего типа, эта

книга знаменует собой важную веху, обозначающую текущее состояние понимания указанной проблемы. По мере того как все больше людей начинают осознавать роль систем программного обеспечения в XXI веке, знание того, как лучше всего реагировать на изменения, сохраняя достигнутое, будет важнейшим навыком в области создания программного обеспечения.

— *Мартин Фаулер*
martinfowler.com
Сентябрь 2017 г.

Введение

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

Моноширинный шрифт

Используется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных, функции, базы данных, типы данных и переменных окружения, инструкции и ключевые слова.



Так выделяются советы и предложения.

От научного редактора перевода

Эта книга как нельзя лучше подойдет новичкам в области архитектуры ПО, которые хотят повысить свою квалификацию. «Эволюционная архитектура» хорошо расширяет кругозор, так как в ней содержится большой объем информации без лишних усложнений.

Начинающего разработчика книга вдохновит на создание архитектуры, которая будет соответствовать концепциям авторов.

Практикующий опытный специалист, в своих проектах уже имевший дело со слоистой архитектурой, возможно, скептически отнесется к идеям, изложенным в этом издании, но через пару месяцев после прочтения переосмыслит свой подход и даже попробует что-то внедрить. А при переходе в новый проект наверняка попытается построить архитектуру с возможностью эволюционирования. Ведь никому не хочется в сотый раз проектировать одно и то же.

Именно такой подход описывают Нил Форд, Ребекка Парсонс и Патрик Куа, чтобы вы захотели и смогли перевести IT-отрасль на эволюционную архитектуру.

Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.

1005 Gravenstein Highway North Sebastopol, CA 95472

800-998-9938 (США или Канада)

707-829-0515 (международный и местный)

707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги <http://oreil.ly/2eY9gT6>.

Свои пожелания и вопросы технического характера отправляйте по адресу bookquestions@oreilly.com.

Дополнительную информацию о книгах, обсуждения, конференции и новости вы найдете на веб-сайте издательства: <http://www.oreilly.com>.

Ищите нас в Facebook: <http://facebook.com/oreilly>.

Следуйте за нами в «Твиттере»: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Дополнительная информация

Авторы ведут сопутствующий сайт этой книги: <http://evolutionaryarchitecture.com>.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Благодарности

Нил хотел бы выразить благодарность всем участникам конференций, на которых он выступал за последние несколько лет и которые помогли ему уточнить и исправить приведенный в книге материал. Он также хотел бы поблагодарить технических рецензентов, которые предоставили превосходную обратную связь и рекомендации. Особая благодарность редакторам Венкату Субраманиуму (Venkat Subramaniam), Оуэну Вудсу (Eoin Woods), Саймону Брауну (Simon

Brown) и Мартину Фаулеру (Martin Fowler). Нил также хотел бы поблагодарить своих котов Уинстона, Паркера и Изабеллу за полезные перерывы, которые всегда приводят к озарениям. Он благодарит своего друга Джона Дресчера (John Drescher) и его коллег из компании *ThoughtWorks*; Нормана Запьяна (Norman Zapien) за его проникательность, его группу *Pasty Geeks* и расположенный по соседству *Cocktail Club* за дружбу и поддержку. И наконец, ему хотелось бы поблагодарить свою многострадальную жену, которая с улыбкой переносит путешествия Нила и другие издержки профессии.

Ребекка хотела бы поблагодарить всех своих коллег, участников конференций и докладчиков, а также авторов, которые на протяжении многих лет генерировали идеи, предоставляли инструменты и методы, задавая уточняющие вопросы об этой области и архитектуре с эволюционным развитием. Вслед за Нилом она выражает благодарность техническим рецензентам за тщательное чтение и комментарии. Ребекке также хотелось бы поблагодарить соавторов за все проливающие свет разговоры и обсуждения в процессе совместной работы над этой книгой. В частности, она благодарна Нилу за полезные обсуждения и даже, может быть, споры, которые велись несколько лет в отношении различий между независимой архитектурой и архитектурой с эволюционным развитием. Эти идеи прошли длинный путь с момента первого разговора.

Патрику хотелось бы поблагодарить всех коллег и клиентов компании *ThoughtWorks*, которые направляли эту работу в нужное русло и предоставили испытательную среду для координации идей в построении архитектуры с эволюционным развитием. Ему бы также хотелось присоединиться к благодарностям Нила и Ребекки техническим рецензентам, которые помогли значительно улучшить эту книгу. И наконец, он хотел бы поблагодарить соавторов за возможность совместной работы над этой книгой, несмотря на большую разницу в часовых поясах и редкие личные встречи.

1

Архитектура программного обеспечения

Долгое время разработчики вели борьбу за краткое определение архитектуры программного обеспечения, так как это достаточно большая и постоянно меняющаяся область. Ральф Джонсон превосходно определил архитектуру программного обеспечения как «важную штуку (чем бы это ни было)». Работа разработчика архитектуры состоит в том, чтобы понять и сбалансировать все эти важные вещи (чем бы они ни были).

Важная часть разработки архитектуры состоит в том, чтобы понять требования той или иной сферы бизнеса или объекта к предлагаемому решению. Хотя эти требования работают как мотивация использования программного обеспечения для решения поставленной задачи, они в конечном счете представляют собой лишь один из факторов, которые разработчики архитектуры (архитекторы) должны учитывать при создании архитектуры. Разработчики архитектуры должны также учитывать многие другие факторы. Некоторые факторы явные (такие, как соглашения об уровне предоставления услуг), а другие неявные для той или иной области бизнеса (например, компания занимается слиянием и поглощением других компаний). Поэтому качество архитектуры программного обеспечения проявляется в способности ее разработчика анализировать бизнес-требования и требования предметной области наряду с другими важными факторами, чтобы найти решение, которое оптимально уравнивало бы все проблемы. Область архитектуры программного обеспечения образуется из объединения всех этих факторов, как это показано на рис. 1.1.



Рис. 1.1. Вся область архитектуры окружена требованиями и различными возможностями

Как видно на рис. 1.1, требования предметной области существуют вместе с другой функциональностью архитектуры, определяемой разработчиком. Сюда входит широкий диапазон внешних факторов, которые могут изменять процесс решения в отношении того, как строить систему программного обеспечения. Для примера проверьте приведенный в табл. 1.1 список.

Таблица 1.1. Частичный перечень возможностей

доступность	идентифицируемость	точность	адаптивность	администрируемость
ценовая доступность	гибкость	контролируемость	автономность	пригодность
совместимость	сочетаемость	конфигурируемость	корректность	достоверность
настраиваемость	способность к разделению	определяемость	демонстрируемость	функциональная надежность
способность к отладке	способность к развертыванию	открытость	распределяемость	продолжительность
эффектность	эффективность	пригодность к использованию	способность наращивания	прозрачность отказов

отказоустойчивость	верность передачи информации	универсальность	возможность контроля	возможность установки целостности
совместимость	обучаемость	восстанавливаемость	управляемость	мобильность
модифицируемость	модульность	работоспособность	независимость	портативность
точность	предсказуемость	возможность оценки	продуктивность	доказуемость
восстанавливаемость	значимость	надежность	повторяемость	воспроизводимость
устойчивость к внешним возмущениям	быстрота реагирования	возможность повторного использования	прочность	безопасность
масштабируемость	незаметность для пользователя	автономность	удобство обслуживания	способность обеспечения безопасности
простота	стабильность	соответствие стандартам	живучесть	устойчивость
легкость сопровождения	способность получения заданных свойств	тестируемость	своевременность	отслеживаемость

При создании программного обеспечения архитектор должен определить самые важные из этих возможностей. Однако многие из этих факторов противодействуют друг другу. Например, достижение высокой производительности и высокой масштабируемости может быть затруднительно, потому что достижение того и другого требует точного баланса архитектуры, операций и многих других факторов. В результате необходим анализ проекта архитектуры, а неизбежное столкновение конкурирующих факторов требует баланса, но балансировка за и против каждого решения в проектировании архитектуры ведет к *компромиссам*, на которые так часто жалуются разработчики архитектуры. В последние несколько лет инкрементная разработка программного обеспечения создала фундамент для переосмысления того, как архитектура меняется со временем, и того, как можно за-

щитить в ходе эволюционного развития важные характеристики архитектуры. Эта книга связывает между собой новый взгляд на *архитектуру и время*.

Нам хотелось бы добавить новую возможность в архитектуру программного обеспечения, а именно *способность к эволюционному развитию*.

Архитектура с эволюционным развитием

Несмотря на все наши усилия, программное обеспечение со временем становится все сложнее менять. По разным причинам те части, которые составляют системы программного обеспечения, не позволяют выполнять простые изменения и становятся со временем все более хрупкими и неподатливыми. Изменения в проектах по разработке программного обеспечения обычно обусловлены повторной оценкой функциональности и/или границами области применения. Изменения другого типа происходят вне области контроля архитекторов и специалистов по долгосрочному планированию. Хотя разработчикам архитектуры нравится чувствовать себя способными к стратегическому планированию, постоянно меняющаяся среда разработки программного обеспечения делает это затруднительным. Поскольку мы не можем избежать изменений, то нам необходимо их использовать.

Как можно осуществлять долгосрочное планирование, если все постоянно меняется?

В биологическом мире окружающая среда непрерывно меняется под воздействием природных и антропогенных причин. Например, в начале 1930-х годов в Австралии были проблемы с жуками-носорогами, которые вредили побегам сахарного тростника и делали его выращивание и сбор невыгодными. В июне 1935 года в ответ на это существовавшее в то время Бюро управления экспериментальными станциями по выращиванию сахарного тростника стало использовать для борьбы с жуками тростниковую жабу, которая раньше населяла

Южную и Среднюю Америку¹. После разведения в неволе некоторое число молодых тростниковых жаб в июле и августе 1935 года было выпущено в Северном Квинсленде. Обладающая ядовитой кожей и не имеющая врагов, тростниковая жаба широко распространилась; на сегодняшний день их число достигло 200 миллионов. Мораль: внесение изменений в высокодинамичную экосистему может привести к непредсказуемым результатам.

Экосистема разработки программного обеспечения включает в себя все рабочие инструменты, фреймворки, библиотеки и практический опыт, накопленный в процессе разработки. Эта экосистема формирует равновесие, во многом напоминающее биологическую систему, которое разработчики могут понимать и строить внутри него. Однако это равновесие динамичное, сопровождающееся непрерывным появлением новых вещей, нарушающих баланс, пока не установится новое равновесное состояние. Представим себе перевозящего коробки моноциклиста: эта система *динамичная*, так как старается сохранять вертикальное положение, и *равновесная*, потому что продолжает поддерживать баланс. В среде разработки программного обеспечения каждое нововведение или новая практика может нарушить статус-кво, вынуждая заново достигать равновесия. Мы продолжаем подбрасывать моноциклисту еще больше коробок, вынуждая его всякий раз восстанавливать равновесие.

Разработчики архитектуры многим напоминают нашего злополучного моноциклиста, непрерывно балансирующего и адаптирующегося к изменяющимся условиям. Практика проектирования непрерывной поставки представляет собой такой тектонический сдвиг в равновесном состоянии: встраивание ранее разрозненных функций (таких, например, как операции) в цикл разработки программного обеспечения открывает новые перспективы на понятие *изменений*. Разработчики архитектуры корпоративных приложений больше не могут опираться на пятилетние планы, потому что вся концепция разработки про-

¹ Clarke, G. M., Gross, S., Matthews, M., Catling, P. C., Baker, B., Hewitt, C. L., Crowther, D., & Saddler, S. R. 2000, Environmental Pest Species in Australia, Australia: State of the Environment, Second Technical Paper Series (Biodiversity), Department of the Environment and Heritage, Canberra.

граммного обеспечения видоизменится за этот период, потенциально превращая долгосрочные решения в категорию спорных.

Разрушительные изменения сложно прогнозировать даже опытным практикам. Появление инструмента композиции контейнеров Docker (<https://www.docker.com/>) является примером непостижимого сдвига в промышленности. Однако мы можем проследить появление контейнеризации с помощью серии небольших последовательных шагов. Некоторое время назад операционные системы, серверы приложений и прочие элементы инфраструктуры были субъектами коммерческой деятельности, требующими лицензирования и больших затрат. Многие архитектуры, проектировавшиеся в эти годы, были ориентированы на эффективное применение совместно используемых ресурсов. Постепенно операционная система Linux стала вполне приемлемой для многих предприятий, и *денежные* затраты на операционные системы сократились до нуля. Затем DevOps начали практическое применение автоматической инициализации машин с помощью таких инструментов, как Puppet (<https://www.puppet.com/>) или Chef (<https://www.chef.io/>), сделав ОС Linux *эксплуатационно* свободной. После того как экосистема стала свободной и нашла широкое применение, стала неизбежной консолидация распространенных форматов, используемых для переноса документов; появился Docker. Но контейнеризация не смогла бы появиться без всех эволюционных шагов, ведущих к этому завершению.

Программные платформы, которые мы используем, служат примером непрерывной эволюции. Новые версии языков программирования предлагают лучшие прикладные программные интерфейсы (API — application programming interfaces) для повышения универсальности или для области применения при решении новых задач; новые языки программирования предлагают разные парадигмы и разные наборы конструирования. Например, язык Java был представлен как замена языка C++, чтобы облегчить написание сетевого кода и улучшить процесс управления памятью. Если посмотреть на прошедшие 20 лет, мы заметим, что многие языки все еще непрерывно развивают API, в то время как у абсолютно новых языков программирования периодиче-

ски возникают новые проблемы. Эволюция языков программирования показана на рис. 1.2.

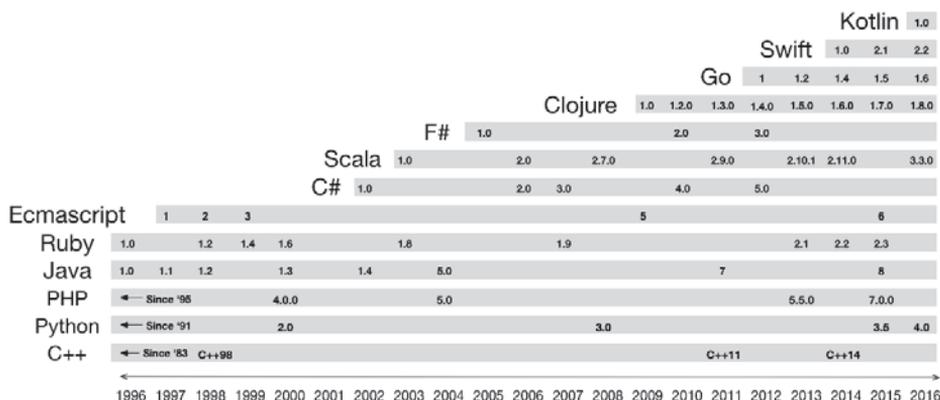


Рис. 1.2. Эволюция популярных языков программирования

Независимо от того, какой именно аспект разработки программного обеспечения мы рассматриваем — программные платформы, языки, операционную среду, технологию поддержки постоянного хранения данных и т. п., — мы ожидаем постоянных изменений. Хотя мы не можем предсказать, когда произойдут изменения в технической области или области применения или какие именно изменения сохранятся, мы знаем, что изменения неизбежны. Следовательно, мы должны разрабатывать архитектуру нашей системы, зная, что изменится техническая область.

Если экосистема непрерывно меняется непредсказуемым образом и прогнозирование изменений невозможно, то что тогда является *альтернативой* жесткому планированию? Архитекторы корпоративных приложений и другие разработчики должны научиться адаптироваться. Частью традиционного обоснования, лежащего в основе долгосрочного планирования, были вопросы финансирования; изменения программного обеспечения были дорогостоящими. Однако современная практика проектирования свела на нет этот фактор, сделав изменения менее затратными путем автоматизации

выполнявшихся вручную в прошлом процессов и в результате появления других практик, таких как DevOps (development и operations).

Многие разработчики микропроцессорных систем столкнулись с тем, что некоторые части их систем было сложнее модифицировать, чем остальные. Вот почему *архитектуру программного обеспечения* определяют как «часть системы, которую сложно будет менять потом». Это удобное определение все делило на те части, которые *можно* изменить без особых усилий, и на те, которые менять действительно сложно. К сожалению, это определение превращается в «белое пятно», когда начинаешь думать об архитектуре: предположение разработчиков, что изменения сложно внести, становится самосбывающимся пророчеством.

Несколько лет назад некоторые разработчики инновационных архитектур программного обеспечения переосмыслили проблему «сложно будет менять позже» в новом свете: что, если возможность менять мы построим *в* архитектуру? Другими словами, если *простота изменения* является базовым принципом архитектуры, тогда вносить изменения будет несложно. Встраивание в архитектуру способности эволюционировать в свою очередь приводит к появлению целого ряда новых характеристик, снова нарушая динамические характеристики.

Даже если экосистема не меняется, то что можно сказать насчет постепенного ухудшения характеристик архитектуры? Разработчики проектируют архитектуры, но затем подвергают их воздействию хаотичного реального мира, *внедряющего* нечто поверх архитектуры. Каким образом разработчики архитектуры защищают важные части, которые они таковыми определили?

Как можно защитить созданную архитектуру от постепенной деградации?

Постепенное ухудшение, которое часто называют *битовой деградацией* (bit rot), возникает во многих организациях. Разработчики архитектуры выбирают определенный паттерн архитектуры для

выполнения требований той или иной области применения и соответствующие возможности, но эти характеристики часто начинают со временем ухудшаться. Например, если архитектор создал много-слойную архитектуру с верхними слоями представления и нижними слоями хранения, а также несколькими промежуточными слоями, то разработчики, работающие над отчетами, из соображений производительности будут часто спрашивать разрешения на прямой доступ к хранящимся в нижнем слое данным из слоя представления, обходя другие слои. Разработчики архитектуры создают слои, чтобы изолировать изменения. Затем разработчики обходят эти слои, увеличивая связанность и нарушая обоснования, которые лежат в основе формирования слоев.

После того как были определены важные характеристики архитектуры, как можно *защитить* эти характеристики, чтобы противодействовать их ухудшению? Добавление *способности к эволюционному развитию* предполагает защиту остальных характеристик по мере эволюции системы. Например, если разработчик архитектуры спроектировал архитектуру, способную масштабироваться, ему не хотелось бы, чтобы характеристики системы ухудшались по мере эволюции системы. Таким образом, *способность к эволюционному развитию* является метахарактеристикой, архитектурной оболочкой, которая защищает все остальные характеристики архитектуры.

В этой книге показано, что побочным эффектом архитектуры с эволюционным развитием являются механизмы, используемые для защиты важных архитектурных характеристик. Мы проанализировали идеи, лежащие в основе *непрерывно меняющейся архитектуры* (continual architecture): построение архитектур, не имеющих конечного состояния, предназначенных для эволюции с постоянно меняющейся экосистемой разработки программного обеспечения и содержащих встроенные средства защиты важных характеристик архитектуры. Мы не пытаемся определить архитектуру программного обеспечения в совокупности; существует много других характеристик (<https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>). Вместо этого мы делаем акцент на расширение используемых в настоящее время определений,

чтобы добавить понятие *времени и изменения* в качестве элементов первого класса.

Приводим наше определение архитектуры с эволюционным развитием:

Архитектура с эволюционным развитием поддерживает управляемые, постепенные и последовательные изменения сразу в нескольких направлениях.

Инкрементные изменения

Инкрементные изменения (Incremental change) описывают два аспекта архитектуры программного обеспечения: то, как команды постепенно формируют программное обеспечение, и то, как они его развертывают.

В процессе разработки архитектура, допускающая небольшие, постепенные изменения, легче эволюционирует, потому что у разработчиков есть небольшие границы изменений. В плане развертывания инкрементное изменение относится к уровню модульности и уменьшению связанности системы для бизнес-функций и тому, как они отображаются в архитектуре. Приведем примеры по порядку.

У крупной компании по продажам виджетов *Penultimate Widgets* есть свой сайт с каталогом виджетов, обеспеченный архитектурой микросервиса и современной практикой проектирования. Одной из особенностей сайта является возможность пользователей оценивать разные виджеты по рейтинговой звездочной системе. Другие услуги, предоставляемые *Penultimate Widgets*, также нуждаются в оценке (оценка клиентских сервисов, транспортных услуг и т. п.), поэтому эти услуги совместно используют систему оценок. Однажды команда, занимающаяся оценками, выпускает новую версию системы, которая позволяет использовать промежуточные категории, что является небольшим, но значимым обновлением. Другие сервисы, для которых система оценок не требует перехода на новую версию, постепенно совершают удобный для них переход. Часть практик DevOps включает в себя контроль архитектуры не только сервисов, но также маршрутов

между ними. Когда команда контроля операций видит, что ни одна из операций не направляется в течение определенного интервала времени в определенный сервис, она автоматически выводит этот сервис из экосистемы.

Это пример инкрементного изменения на уровне архитектуры: исходный сервис может работать вместе с новым, если в нем нуждаются другие сервисы. Команды могут переходить на новое поведение во время перерывов в работе (или по мере необходимости), а старая версия автоматически отбрасывается.

Для успешного осуществления инкрементного изменения требуется координация, основанная на использовании практики непрерывной поставки. Целиком эта практика нужна не во всех случаях, но достаточно часто используется в естественной среде. Мы поясним, как можно достичь инкрементного изменения, в главе 3.

Управляемое изменение

После того как архитекторы выбрали важные характеристики, им необходимо *направлять* изменения в архитектуре, чтобы защитить эти характеристики. С этой целью нами из вычислений с помощью эволюционных алгоритмов была заимствована концепция *функций пригодности* (fitness functions). Функция пригодности является целевой функцией, используемой для оценки того, как близко решение перспективного проектирования к достижению поставленных целей. В вычислениях с помощью эволюционных алгоритмов функция пригодности определяет, изменился ли со временем алгоритм в лучшую сторону. Другими словами, по мере создания каждого варианта алгоритма функция пригодности определяет, насколько пригоден каждый вариант, основываясь на том, как разработчик этого алгоритма определяет пригодность.

Мы ставим аналогичную цель в архитектуре с эволюционным развитием — по мере эволюционирования архитектуры нам нужен механизм оценки того, как изменения влияют на важные характеристики архитектуры и предотвращают их ухудшение со временем. Понятие

функции пригодности охватывает различные механизмы, которые мы используем для гарантии того, что архитектура не меняется нежелательным образом, включая определенные показатели, тесты и другие инструменты проверки. После того как разработчик определил характеристики архитектуры, которые необходимо защищать по мере развития событий, он задает одну или несколько функций пригодности для защиты этих характеристик.

Исторически сложилось так, что определенную часть архитектуры рассматривают как управляющую, и разработчики совсем недавно приняли идею, позволяющую осуществлять изменения с помощью самой архитектуры. Функции архитектурной пригодности позволяют принимать решения в контексте потребностей организации и управления предприятием, одновременно создавая контролируемую основу для этих решений. Архитектура с эволюционным развитием не является не имеющим ограничений, ненадежным методом разработки программного обеспечения. Напротив, этот метод уравнивает необходимость в быстром изменении и сохранении находящихся вокруг систем и характеристик системы. Функция пригодности приводит в действие архитектурные решения, управляя архитектурой, одновременно позволяя выполнять изменения, необходимые для поддержки деловой и технологической среды.

Мы используем *функции пригодности* для создания указаний по эволюции архитектуры; подробно рассмотрим их в главе 2.

Многочисленные области архитектуры

В мире нет отдельных систем. Мир непрерывен. Место, где следует проводить границу вокруг системы, зависит от цели проводимого исследования.

— *Донелла Медоуз (Donella H. Meadows)*

Классическая физика постепенно научилась анализировать Вселенную, основываясь на точках опоры, что привело к появлению

классической механики. Однако появление более точных приборов и открытие сложных явлений в начале XX века постепенно привели к понятию относительности. Ученые осознали, что все, что раньше рассматривалось как отдельные явления, на самом деле связано друг с другом. Начиная с 1990-х годов, свободные от предрассудков разработчики архитектуры стали все в большей степени рассматривать архитектуру программного обеспечения как многомерную систему. Непрерывная поставка расширила эту точку зрения до включения операций. Однако разработчики архитектуры часто акцентировали свое внимание на *технической* архитектуре, но это лишь одна из областей разработки программного обеспечения. Если разработчик хотел создать архитектуру, которая могла бы эволюционировать, он должен был учитывать все части системы, в которой меняются взаимодействия. Мы хорошо знаем из физики, что все относительно. И архитекторы знают, что проект программного обеспечения многомерен.

Чтобы построить способную к эволюции систему программного обеспечения, архитекторы должны продумать не только техническую архитектуру. Например, если проект включает в себя реляционную базу данных, то структура и связь между элементами базы данных будут также эволюционировать со временем. Разработчикам также не хотелось бы строить систему, которая в результате эволюции имела бы уязвимости системы безопасности. Здесь приводятся примеры *областей* архитектуры, а именно тех ее частей, которые соответствуют друг другу принципиально по-новому. Некоторые размерности часто укладываются в то, что принято называть *архитектурными особенностями* (см. приведенный выше перечень возможностей), но *области* фактически шире, охватывая элементы, традиционно находящиеся за пределами технической архитектуры. В каждом проекте есть область, которую разработчик должен учитывать при планировании эволюционного развития. Далее приводятся некоторые распространенные области, которые влияют на способность к эволюции в современных архитектурах программного обеспечения:

Техническая

Реализация определенных частей архитектуры: фреймворков, зависимых библиотек и реализации языка(-ов).

Данные

Схемы баз данных, макеты таблиц, планирование оптимизации и т. п. Администратор базы данных обычно имеет дело с этим типом архитектуры.

Система обеспечения безопасности

Определяет принципы безопасности, инструкции и инструмент для оказания помощи в выявлении недостатка ресурсов.

Операционная/системная

Особенность, связанная с тем, как архитектура сопоставляется с существующей физической или виртуальной инфраструктурой: серверами, кластерами машин, коммутаторами, облачными хранилищами и т. п.

Каждая из этих перспектив образует *область* архитектуры, представляющую собой определенным образом выделенные части, соответствующие той или иной перспективе. Наша концепция архитектурных областей включает традиционные архитектурные характеристики (возможности) и любой другой элемент, который вносит вклад в построение программного обеспечения. Каждая из этих форм представляет собой ту или иную точку зрения на архитектуру, которую мы хотим сохранить по мере эволюции проблемы и изменения окружающего нас мира.

Существуют разнообразные методы концептуального деления архитектуры на части. Например, модель представления архитектуры «4 + 1», которая фокусируется на разных перспективах, исходя из различных ролей, и которая использовалась Институтом инженеров электротехники и электроники (IEEE) в определении архитектуры программного обеспечения, делит экосистему на такие представления, как *логическое, процессное, физическое* и *представление уровня разработки*. В книге *Software Systems Architecture* (<https://www.viewpoints-and-perspectives.info/>) авторы представили перечень точек зрения на архитектуру программного обеспечения, охватывающий

большой набор ролей. Аналогичным образом, модель С4 Саймона Брауна делит на части функциональность, связанную с концептуальной организацией. В противоположность этому, в нашей книге мы не пытаемся создать систему классификации измерений, а выявляем те из них, которые используются в существующих проектах. Независимо от категории, в которой находится та или иная функциональность, разработчик архитектуры должен защищать эту область. Разные проекты имеют различную функциональность, что приводит к неповторяющимся наборам измерений для разных проектов. Любой из этих методов дает полезную аналитическую оценку, особенно новых проектов, но в случае существующих проектов необходимо иметь дело с тем, что есть.

Когда архитекторы мыслят в терминах областей архитектуры, это дает механизм, с помощью которого они могут анализировать способность эволюционного развития различных архитектур, оценивая то, как каждая важная область реагирует на изменение. По мере того как системы становятся все более тесно связанными с конкурирующими функциональностями (масштабируемостью, безопасностью, распределением, транзакциями и т. п.), разработчики должны расширять те области, которые они используют в проектах. Для построения эволюционирующей системы архитекторы должны планировать то, как система могла бы эволюционировать в каждой из ее важных областей.

Вся архитектурная область проекта состоит из требований к программному обеспечению и других областей. Мы может применять функции пригодности для защиты характеристик архитектуры по мере совместного эволюционного развития во времени архитектуры и ее экосистемы, как это показано на рис. 1.3.

На рис. 1.3 представлены архитектуры с выявленными важными для этой предметной области возможностями, такими как *контролируемость, данные, безопасность, производительность, доступность* и *масштабируемость*. Поскольку со временем требования предметной области эволюционируют, каждая характеристика архитектуры использует функции пригодности для защиты их целостности.

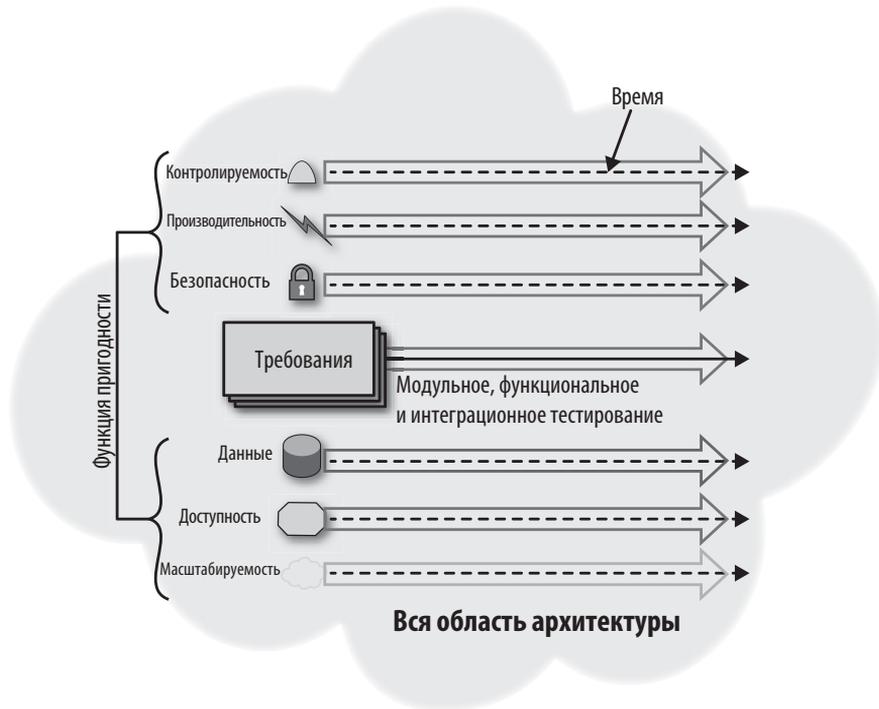


Рис. 1.3. Любая архитектура состоит из требований и других измерений, каждое из которых защищено функциями пригодности

В то время как авторы этой книги подчеркивают важность целостного подхода при разработке архитектуры, следует также понимать, что большая часть эволюционирующей архитектуры относится к паттернам технической архитектуры и сопутствующим темам, таким как связанность (coupling) и сцепление (cohesion). В главе 4 мы рассмотрим, как связанность технической архитектуры влияет на способность эволюционировать, а в главе 5 рассмотрим влияние на связанность данных.

Связанность применима не только к структурным элементам в проектах разработки программного обеспечения. Многие компании, занимающиеся разработкой программного обеспечения, недавно обнаружили влияние структуры команды на архитектуру. Мы рассмотрим все аспекты связанности в программном обеспечении, однако влияние

команды возникает на ранних этапах и так часто, что мы должны пояснить это прямо сейчас.

Закон Конвея

В апреле 1968 года Мелвин Конвей представил статью в журнал *Harvard Business Review* под названием «Как комитеты изобретают?» (*How Do Committees Invent*). В этой статье Конвей высказывает мысль, что социальные структуры, в частности каналы связи между людьми, неизбежным образом влияют на проект конечного продукта.

Как описывает Конвей, на ранней стадии каждого проекта для того, чтобы понять, как разбить зоны ответственности на различные структуры, делается попытка понять систему на высоком уровне. То, как команда разбивает проблему на части, влияет на выбор, который они должны будут сделать позже. Конвей приводит в систему то, что теперь известно как *закон Конвея*:

Организации, проектирующие системы, ограничены дизайном, который копирует структуру коммуникации в этой организации.

— *Мелвин Конвей (Melvin E. Conway)*

Как заметил Конвей, когда инженеры разбивают крупные задачи на группу мелких задач для передачи части работ другим, возникают проблемы координации работ. Во многих организациях формальные структуры коммуникации или жесткая иерархия решают эту проблему координации, но это часто приводит к негибким решениям. Например, в слоистой архитектуре, где команда разделена на части технической функциональностью архитектуры (пользовательский интерфейс, предметная логика и т. п.), которая решает общие проблемы и которая делит архитектуру вертикально через все слои, это повышает неэффективность координации. Люди, которым пришлось работать в стартапах, а затем присоединиться к крупной мультинациональной

корпорации, вероятно, почувствовали разницу между гибкой, адаптивной культурой первой и негибкими структурами коммуникации последней. Хорошим примером закона Конвея в действии может служить попытка изменить контракт, заключенный между двумя сервисами, которая может быть осложнена тем, что успешное изменение сервиса одной командой требует координированных и согласованных усилий другой.

В этой статье Конвей предупреждает разработчиков систем программного обеспечения обращать внимание не только на архитектуру и проект программного обеспечения, но также на передачу части работ и координацию работы между командами.

Во многих организациях команды разделены в соответствии с их функциональными навыками. Далее приведены некоторые общеизвестные примеры:

Фронтенд-разработчики

Команда со специализированными навыками в области пользовательских интерфейсов (UI, User Interface) (например, HTML, мобильных устройств, настольных ПК).

Бэкенд-разработчики

Команда с уникальными навыками создания сервисов серверной части, иногда интерфейсов для прикладных программ (API, Application Program Interface).

Разработчики баз данных

Команда с уникальными навыками в построении хранилищ данных и логических сервисов.

В организациях с функциональными «анклавами», которые существуют в полном отрыве от других участков, руководство разделяет команды в угоду отделу персонала, не обращая особого внимания на инженерную эффективность. Несмотря на то что каждая команда может хорошо справляться со своей частью проекта (например, с построением экрана, добавлением интерфейса для прикладных

программ или сервисами в серверной части системы, с разработкой нового механизма хранения данных), для того чтобы создать новую бизнес-возможность или функцию, все эти команды должны работать совместно. Команды обычно оптимизируют эффективность для выполнения срочных задач, а не для достижения более абстрактных, стратегических бизнес-целей, особенно находясь под давлением графика. Вместо сквозной разработки функции команды часто направляют свои усилия на разработку компонентов, которые могут или не могут хорошо работать друг с другом.

В таком организационном разделении функция, зависящая от всех трех команд, занимает больше времени, поскольку каждая команда работает над своим компонентом в разное время. Например, рассмотрим распространенное в бизнес-среде обновление страницы каталога. Это изменение ведет за собой изменения пользовательского интерфейса, бизнес-правил и схемы базы данных. Если каждая команда работает в своем собственном «анклаве», они должны координировать графики работ, увеличивая время, необходимое для реализации разрабатываемой функции. Это удачный пример того, как структура команд может влиять на архитектуру и способность эволюционировать.

Как отмечает в этой статье Конвей, *всякий раз при делегировании работ чей-то круг обязанностей сужается, при этом также сужается класс допустимых вариантов проектирования*. Иначе говоря, сложно что-то менять, если это что-то принадлежит кому-то еще. Архитекторы должны обращать внимание на то, как работа делится и передается в группы для согласования целей архитектуры со структурой команд.

Многие компании, занимающиеся разработкой архитектур, таких как архитектура микросервисов, располагают команды вокруг границ сервисов, а не используют их как разрозненные технические части архитектуры. В издании *Technology Radar* (<https://www.thoughtworks.com/radar>) нашей компании *ThoughtWorks* мы называли этот подход обратным маневром Конвея. Организация команд таким образом идеально подходит потому, что структура групп будет влиять на множество областей при разработке программного обеспечения и отражать раз-

мер и границы проблемы. Например, при построении архитектуры микросервисов компании обычно группируют команды так, чтобы это напоминало разрабатываемую архитектуру, сокращая функциональные анклавов и включая участников команд, которые охватывают все бизнес-аспекты и технические аспекты архитектуры.



Структура команды должна выглядеть как целевая архитектура. Тогда ее будет проще достичь.

ЗНАКОМСТВО С PENULTIMATEWIDGETS И С ИХ ОБРАТНЫМ ЗАКОНОМ КОНВЕЯ

В книге мы используем примеры компании PenultimateWidgets «Продавца каждого, от первого до последнего, виджета» (и других «штучек»). Компания постепенно обновляет большую часть своей IT-инфраструктуры. У них есть унаследованные системы, которые компания хочет сохранить на протяжении какого-то времени, а также новые стратегические системы, которые требуют более итеративных подходов. В этих главах мы выделим проблемы и укажем решения, которые были разработаны PenultimateWidgets.

Первое замечание, сделанное их архитекторами, касалось команд разработчиков программного обеспечения. В прежних монолитных приложениях использовалась слоистая архитектура для разделения слоев представления, логики предметной области, хранения данных и операций. Команды отражают эти функции: все разработчики UI находятся вместе в одном помещении, разработчики и администраторы баз данных находятся в «анклаве», а оперативные процессы выполняются третьей стороной удаленно.

Когда разработчики начинают работу над новыми элементами архитектуры, архитектурой микросервисов с их собственными точными сервисами, затраты на координацию взлетают до небес. Поскольку эти сервисы были построены вокруг областей (таких, как *CustomerCheckout*), то в отличие от технической архитектуры изменение одной области приводит к необходимости определенной координации между всеми «анклавами».

Вместо этого, используя закон Конвея в PenultimateWidgets, создаются межфункциональные команды, которые согласуют области применения сервисов: команда каждого сервиса состоит из владельца сервиса, нескольких разработчиков, бизнес-аналитика, администратора базы данных, специалистов по обеспечению качества и выполнению операций.

Влияние команды показано в разных главах книги с примерами того, какие значительные последствия это имеет.

Почему эволюционное развитие?

Распространенный вопрос об эволюционном развитии архитектуры кроется в названии: почему мы называем архитектуру *эволюционирующей*, а не как-то иначе? К одному из возможных названий относится *инкрементная, непрерывная, гибкая, реактивная и независимая*. Но каждый из этих терминов что-то упускает. Определение архитектуры с эволюционным развитием, которое мы здесь приводим, содержит две важные характеристики: инкрементное и управляемое.

Термины *непрерывная, гибкая и независимая* передают идею изменения во времени, которое определено является важной характеристикой архитектуры с эволюционным развитием, но ни один из этих терминов в явном виде не передает мысль о том, *как* архитектура меняется или каким должно быть ее конечное состояние. В то время как все термины предполагают изменение среды, ни один из них не указывает на то, как должна выглядеть архитектура. «*Управляемая*» в нашем определении отражает то, какой мы хотим получить архитектуру, то есть нашу конечную цель.

Нам кажется предпочтительнее использовать термин *эволюционная*, а не *адаптивная*, потому что мы заинтересованы в архитектуре, которая претерпевает фундаментальное эволюционное изменение, а не то, которое было включено и адаптировано во все более непонятную и ненужную сложность. *Адаптивность* подразумевает поиск способа заставить что-то работать, независимо от изящности или долговеч-

ности найденного решения. Чтобы строить архитектуры, которые по-настоящему развиваются, архитекторы должны поддерживать подлинное изменение, а не решения на скорую руку. Возвращаясь назад к нашей биологической метафоре, можно сказать, что *эволюционное развитие* относится к системе, которая пригодна для определенной цели и может выжить в непрерывно меняющейся среде, в которой она работает. Система может обладать индивидуальной адаптацией, но, как разработчики, мы должны заботиться о способной к эволюции системе в целом.

Краткие выводы

Архитектура с эволюционным развитием включает три основных аспекта: инкрементное изменение, функции пригодности и надлежащую связанность. Далее мы обсудим каждый из этих факторов по отдельности, затем соберем их вместе, чтобы рассмотреть, что потребуется для создания и сохранения архитектуры, поддерживающей постоянное изменение.

2

Функции пригодности

Архитектура с эволюционным развитием поддерживает *управляемые*, инкрементные изменения в различных областях.

— *Наше определение*

Как уже было отмечено, слово *управляемое* указывает на то, что существует некоторая цель, к которой архитектура должна двигаться или демонстрировать такое движение. Авторы позаимствовали концепцию функции пригодности из вычислений с помощью эволюционных алгоритмов, используемую в генетических алгоритмах для определения успеха. Вычисления с помощью эволюционных алгоритмов содержат ряд механизмов, которые позволяют решению постепенно проявляться посредством небольших изменений в каждом поколении программного обеспечения. В каждом поколении решения инженер оценивает текущее состояние: оно ближе к конечной цели или удаляется от нее? Например, при использовании генетического алгоритма для оптимизации конструкции крыла функция пригодности оценивает сопротивление крыла, его вес, поток воздуха и другие характеристики, необходимые хорошей конструкции. Разработчики архитектуры определяют функцию пригодности, чтобы указать, что лучше, а также иметь возможность оценки близости цели. В программном обеспечении функции пригодности проверяют, сохраняются ли выбранные разработчиком важные характеристики.

Мы используем эту концепцию для определения функции пригодности в архитектуре:

Архитектурная функция пригодности предоставляет оценку целостности некоторых архитектурных характеристик.

— *Наше определение*

Функция пригодности защищает различные архитектурные характеристики, необходимые для системы. Специальные архитектурные требования сильно отличаются в разных системах и организациях, что связано с предметной областью, техническими возможностями и множеством других факторов. Некоторые системы нуждаются в обеспечении высокого уровня безопасности; другим необходима значительная пропускная способность или малое время ожидания. В то же время другие системы должны быть более стойкими к сбоям. Эти соображения формируют функциональные возможности, которые разработчики архитектуры должны контролировать. Концептуально, архитектурная функция пригодности включает в себя механизм защиты функциональных возможностей той или иной системы.

Мы можем также предусмотреть *функцию пригодности всей системы*, представляющую собой набор функций пригодности, каждая из которых соответствует одному или нескольким областям архитектуры программного обеспечения. Применение функции пригодности всей системы способствует нашему пониманию необходимых компромиссов, когда отдельные элементы функции пригодности конфликтуют друг с другом. Так же как в случае задачи многофункциональной оптимизации, может оказаться невозможно оптимизировать все величины одновременно. А это приводит нас к необходимости сделать определенный выбор. Например, в случае архитектурных функций пригодности производительность системы может конфликтовать с ее безопасностью, что может быть связано с затратами на шифрование. В этом классическом примере проклятием разработчиков архитекту-

ры всегда являются *компромиссы* (tradeoff). Компромиссы занимают лидирующую позицию среди причин головных болей разработчиков в их стремлении примирить противоборствующие силы, такие, например, как масштабируемость и производительность. Однако у разработчиков есть еще вечная проблема сравнения этих разных характеристик, поскольку они фундаментально различаются (как яблоко и апельсин) и все заинтересованные стороны уверены, что их функциональность главенствующая. Функция пригодности всей системы дает возможность разработчику продумать расходящиеся функциональности, используя один и тот же объединяющий механизм функций пригодности, выявляющий и сохраняющий важные характеристики архитектуры. Связь между функцией всей системы с ее составляющими функциями пригодности показана на рис. 2.1.



Рис. 2.1. Связь функции всей системы с отдельными функциями пригодности

Функция пригодности всей системы имеет решающее значение в способности архитектуры эволюционировать, поскольку нам нужна основа, которая позволила бы разработчикам сравнивать и оценивать

характеристики архитектуры друг относительно друга. В отличие от более направленных функций пригодности, разработчики архитектуры никогда не будут «оценивать» эту функцию всей системы. Вместо этого она предоставляет указания по приоритезации решений в отношении этой архитектуры в будущем.

Система никогда не является суммой ее частей. Она является продуктом взаимодействий ее частей.

— *Д-р Рассел Л. Акофф (Dr. Russel Ackoff)*

Без управления архитектура с эволюционным развитием становится просто реагирующей на внешние воздействия архитектурой. Таким образом, важные ранние решения для архитектуры любой системы состоят в выявлении важных размерностей, таких как масштабируемость, производительность, безопасность, схемы доступа к данным и т. п. Концептуально это дает возможность разработчикам оценивать важность функции пригодности, основываясь на ее возможности влиять на поведение всей системы.

Сначала функции пригодности задаются строго, а затем проверяется, как они управляют эволюцией архитектуры системы.

Что собой представляет функция пригодности?

С точки зрения математики функция берет на входе данные из некоторого допустимого набора входных значений и создает выходные данные в некотором допустимом наборе выходных значений. В случае программного обеспечения мы также обычно используем термин *функция* для обозначения того, что можно фактически реализовать. Однако, как в случае с критерием приемлемости при разработке гибкого программного обеспечения, функции пригодности для архитектуры с эволюционным развитием могут оказаться не реализуемыми в программном обеспечении (например, требуется ручной процесс в це-

лях регулирования), но разработчики должны при этом определить функции пригодности вручную, чтобы помочь управлять эволюцией системы. Несмотря на то что предпочтительны автоматизированные проверки, некоторые проекты не могут автоматизировать все функции проверки пригодности. Поэтому для разработчиков архитектуры по многим причинам остается полезным устанавливать проверки характеристик архитектуры с помощью функций пригодности, что скоро станет очевидным.

Как пояснялось в главе 1, реальные архитектуры состоят из разных областей, включая требования, предъявляемые к производительности, надежности, безопасности, работоспособности, требования стандартов кодирования и целостности. Нам хотелось бы, чтобы функция пригодности представляла каждое из требований, предъявляемых к архитектуре. Разработчики обычно выражают функции пригодности, используя разные типы механизмов, такие как тесты или системы показателей. Рассмотрим несколько примеров, а затем более подробно разные типы функций.

Требования к производительности наиболее эффективно используют функции пригодности. Рассмотрим требование: все сервисы должны откликаться в течение 100 мс. Мы можем использовать тест (то есть функцию пригодности), который измеряет время отклика на запрос, получаемый сервисом, и который считается не пройденным, если результат измерения превышает 100 мс. С этой целью каждый новый сервис должен иметь соответствующий тест производительности, добавленный в набор. Разработчики, составляющие тесты, должны решить, какой уровень полноты диапазона и типов входных данных заслуживает доверия при успешном прохождении теста. Они также должны принять решение, когда запускать эти тесты и что делать, если тест оказывается неудачным. Проверка производительности должна проводиться на ранней стадии и достаточно часто, особенно при установлении точек перегиба, когда производительность сильно меняется (обычно в худшую сторону) из-за обновления кода.

Функция пригодности может также использоваться для поддержания стандартов кодирования. Обычно показателем кода является

цикломатическая сложность¹, то есть мера сложности функции или метода. Разработчик может установить пороговое значение для верхней величины, контролируемой модульным тестом, выполняемым в непрерывном режиме, используя один из многих инструментов для оценки этого показателя. В предыдущем примере разработчики принимают решение, когда запустить функцию пригодности для оценки производительности системы. Для выполнения стандартов кодирования разработчикам хотелось бы, чтобы нарушения при составлении обнаруживались немедленно и возникающие проблемы активно устранялись.

Несмотря на необходимость, разработчики не могут всегда полностью использовать функции пригодности из-за их сложности или наличия других ограничений. Рассмотрим случай обхода при сбоях базы данных в результате аппаратного сбоя. В то время как самовосстановление после сбоя может быть полностью автоматизированным (и оно должно быть таким), запуск самого теста лучше всего выполнять вручную. Кроме того, может оказаться намного эффективнее определять успешность теста вручную, хотя скрипты и автоматизация все еще приветствуются.

Эти примеры показывают многочисленное разнообразие форм, которые могут принимать функции пригодности, немедленный отклик на сбой функции пригодности и даже то, когда и как разработчики могут их запускать. Несмотря на то что мы не можем запустить один скрипт и сказать, что «наша архитектура в данный момент имеет суммарную пригодность в 42», мы можем точно и недвусмысленно рассуждать о состоянии архитектуры по функции общей пригодности системы. Мы также можем судить об изменениях, которые могли произойти с пригодностью архитектуры.

И наконец, когда мы говорим, что архитектура с эволюционным развитием управляется функцией пригодности, это значит, что выполняется оценка выборов отдельных изменений архитектуры и всей системы в целом для определения влияния этого изменения. Функции пригодности в совокупности означают то, что важно для нас в используемой

¹ https://ru.wikipedia.org/wiki/Цикломатическая_сложность

архитектуре. Это позволяет принимать решения о компромиссах, что одновременно является значимым и вызывающим беспокойство во время разработки систем программного обеспечения.

Функции пригодности объединяют многие существующие концепции в один механизм, позволяющий разработчикам учитывать стандартным образом многие существующие (часто для данного случая) «нефункциональные требования» к испытаниям. Совокупность важных архитектурных порогов и требований в виде функций пригодности дает возможность более конкретного представления ранее размытых, субъективных критериев оценки. Мы использовали большое число существующих механизмов построения функции пригодности, включая традиционное тестирование, мониторинг и прочие инструменты. Не все тесты относятся к функциям пригодности, но некоторые можно отнести к ним; если тест помогает проверить целостность функциональностей архитектуры, мы можем считать его функцией пригодности.

Категории

Функции пригодности существуют в различных категориях, связанных с их областью применения, частотой, динамикой и другими факторами, включая комбинации категорий там, где это необходимо.

Атомарная и комплексная функции

Атомарная (atomic) функция пригодности используется в единичном контексте и затрагивает только один определенный аспект архитектуры. Удачным примером атомарной функции пригодности является модульное тестирование, которое проверяет некоторые характеристики архитектуры, такие как связанность модулей (мы представим пример функции пригодности этого типа в главе 4). Таким образом, некоторые тесты уровня приложений попадают в категорию функций пригодности, но не все модульные тесты являются функциями пригодности, а только те, которые проверяют характеристики архитектуры.

Некоторые характеристики архитектуры разработчики должны тестировать больше, чем каждую ее область по отдельности. *Комплексные* (holistic) функции пригодности используются в общем контексте и затрагивают комбинации аспектов архитектуры, таких как безопасность и масштабируемость. Разработчики проектируют комплексные функции пригодности с целью того, что объединенные характеристики, которые продолжают работать по отдельности, продолжают функционировать в разных сочетаниях в реальном мире. Например, представим архитектуру, имеющую функции пригодности как для масштабируемости, так и для безопасности. Одним из ключевых элементов, которые проверяет функция пригодности безопасности, являются «просроченные» данные, а ключевым элементом для проверок масштабируемости является текущее число пользователей в пределах определенного диапазона времени задержки. Для обеспечения масштабируемости разработчики применяют кэширование, которое дает возможность масштабировать атомарные функции пригодности. Когда кэширование не подключено, функция пригодности безопасности успешно проходит проверку. Однако при комплексной проверке кэширование делает данные устаревшими, не позволяя им пройти проверку функцией пригодности по безопасности, и комплексный тест пройти не удастся.

Очевидно, что мы не можем протестировать каждую возможную комбинацию элементов архитектуры, поэтому разработчики выборочно используют комплексную функцию пригодности для проверки важных взаимодействий. Эта селективность и приоритетность всегда позволяют разработчикам оценивать сложность реализации определенного сценария проверки (с помощью функций пригодности), тем самым давая возможность оценки значимости той или иной характеристики. Часто взаимодействие между функциональностями архитектуры определяет качество архитектуры, которое оценивается с помощью комплексной функции пригодности.

Триггерные и непрерывные функции

Порядок выполнения является еще одной отличительной чертой функции пригодности. *Триггерные* (triggered) функции пригодности

запускаются при определенных событиях, таких как выполнение модульного теста, развертывание конвейера для выполнения модульных тестов или выполнение специалистом по обеспечению качества исследовательского тестирования. Эти события включают в себя традиционные тесты, такие как модульные, функциональные, разработку через реализацию поведения (BDD — behavior-driven development) и другие тесты, проводимые разработчиками.

Непрерывные (continual) тесты не выполняются по графику, но вместо этого проводится непрерывная проверка таких параметров архитектуры, как скорость транзакций. Например, рассмотрим архитектуру микросервисов, в которой разработчики хотели бы построить функцию пригодности для проверки времени транзакций, то есть сколько времени занимает в среднем транзакция? Построение любого типа триггерного теста обеспечивает скудную информацию о поведении системы в реальном мире. Таким образом, вместо использования запускаемого теста разработчики построили функцию пригодности, которая имитирует транзакцию в производстве при одновременном выполнении всех остальных транзакций. Это позволяет разработчикам проверить поведение системы и собрать о ней реальные данные «в естественной среде».

Разработка, управляемая моделями (MDD — Monitoring-driven development) представляет другую технику тестирования, набирающую популярность. Вместо того чтобы надеяться только на тесты для проверки результативности системы, MDD использует систему контроля для оценки технического состояния и предметной области. Эти непрерывные функции пригодности более динамичны, чем стандартные триггерные тесты.

Статические и динамические функции

Статические (static) функции пригодности имеют неизменные результаты, такие как двоичные *прошел/провалил*, которые дают модульные тесты. Этот тип включает в себя любую функцию пригодности, которая имеет предварительно заданное требуемое значение: двоичное, диапазон числа, включение множества и т. п. В качестве функций

пригодности часто используют метрики. Например, разработчик архитектуры может задать приемлемый диапазон для средней цикломатической сложности методов в базе кода с градациями проверок, используя инструмент метрик, встроенный в систему развертывания конвейера.

Динамические (dynamic) функции пригодности опираются на определение смещения, основанного на дополнительном контексте. Некоторые значения могут зависеть от обстоятельств, и большинство разработчиков архитектуры считают допустимым низкий показатель производительности при работе с крупным масштабом. Например, компания может использовать скользящее значение показателя производительности на основе масштаба, при этом при большом масштабе допускается низкая производительность, но только в пределах допустимых значений.

Автоматизированная и ручная функции

Понятно, что разработчики любят использовать автоматизированные операции, и часть инкрементных изменений включают автоматизированно, что будет подробно описано в главе 3. Неудивительно, что разработчики будут выполнять большую часть функций пригодности в автоматизированном режиме: непрерывная интеграция, развертывание конвейеров и т. п. Действительно, разработчики и специалисты DevOps выполнили колоссальный объем работ с помощью системы непрерывной поставки для автоматизации многих частей экосистемы разработки программного обеспечения, что раньше считалось невозможным. Эта тенденция должна быть сохранена.

Однако при автоматизации всех аспектов разработки некоторые из аспектов программного обеспечения оказывают сопротивление автоматизации. Иногда важная область системы, например такая, как юридические требования, не поддается автоматизации. Например, разработчики приложений в некоторых областях задач должны иметь возможность замены по законодательным причинам сертификата программного обеспечения, что невозможно выполнить автома-

тически. Аналогичным образом проект может стремиться стать более эволюционирующим, но при этом не иметь надлежащей практики проектирования на месте. Например, вероятно, что большая часть систем обеспечения качества все еще функционирует в ручном режиме, и они должны оставаться в этом статусе в ближайшем будущем. В обоих этих случаях (и многих других) нам необходимы функционирующие в *ручном режиме* функции пригодности, которые проверяются человеком.

Понятно, что путь к повышению эффективности ограничивается выполняемыми вручную шагами, при этом многие проекты все еще требуют выполняемых вручную процедур. Мы по-прежнему задаем функции пригодности для этих характеристик и проверяем их, используя выполняемые вручную шаги, например, при развертывании конвейеров (это будет подробно описано в главе 3).

Временная функция

В то время как большинство функций пригодности запускается при изменениях системы, разработчикам хотелось бы встроить компонент времени в процедуру оценки пригодности. Например, если проект использует библиотеку шифрования, разработчик может хотеть создать временную функцию пригодности как напоминание о необходимости проверки того, были ли обновлены важные компоненты. Другим распространенным применением этого типа функции пригодности является тест *повреждений при обновлении* (break upon upgrade). На таких платформах, как «Ruby on Rails», некоторые разработчики не могут дождаться новых функций, входящих в следующий релиз, поэтому они добавляют функцию в текущую версию с помощью *бэкпорта* (back porting), используя самостоятельно написанную реализацию будущей функции. Проблемы возникают, когда проект окончательно обновляется до новой версии, потому что бэкпорт часто несовместим с «реальной» версией. Разработчики используют тесты на *повреждение при обновлении*, чтобы вернуть реализованные с помощью бэкпортирования функции при обновлении.

Функция с преднамеренным развитием

В то время как разработчики будут определять большую часть функций пригодности в начале проекта по мере выяснения роли характеристик архитектуры, некоторые функции пригодности будут появляться в ходе разработки системы. Разработчики никогда не знают в начале все важные части архитектуры (классическую проблему *неизвестного неизвестного* рассмотрим в главе 6) и поэтому должны выявлять функции пригодности по мере эволюции системы.

Предметно-ориентированная функция

Некоторые архитектуры имеют специфические проблемы, связанные с особыми требованиями по безопасности или нормативными требованиями. Например, компания, которая занимается международными переводами средств, может разработать конкретную, непрерывную целостную функцию пригодности, которая проводит тесты безопасности, смоделированные после того, как инструмент Simian Army (описано в главе 3) пытается вывести из строя инфраструктуру. Многие проблемы предметных областей содержат драйверы, которые ведут архитектуру к одному или нескольким наборам важных характеристик. Архитекторы должны использовать эти драйверы в качестве функций пригодности для гарантии того, что важные характеристики архитектуры не будут ухудшаться со временем.

В главе 3 мы приведем примеры объединения этих областей, когда придет время оценивать функции пригодности.

Ранняя идентификация функций пригодности

Команды должны идентифицировать функции пригодности как часть их начального понимания общей функциональности архитектуры, которую их проект должен поддерживать. Они также должны на ранней стадии идентифицировать функцию пригодности своей *системы*, чтобы определить те изменения, которые следует поддерживать. Срав-

нения важности и сложности реализации различных характеристик архитектуры (наряду с функциями пригодности) помогает на ранней стадии расставить приоритеты связанных с риском работ для понимания того, как проектировать с учетом изменений.

Команды, которые не идентифицируют свои функции пригодности, сталкиваются со следующими рисками:

- Неправильные варианты дизайна, которые в конечном итоге приводят к созданию программного обеспечения, не работающему в своей среде.
- Выбор дизайна, который стоит времени и/или денег, но в итоге не подходит.
- Система не способна эволюционировать при изменении среды.

Для каждой системы программного обеспечения команды должны сфокусироваться на подборе и определении наиболее важных функций пригодности. Подбор функций пригодности на ранней стадии помогает разработчикам планировать разделение крупной системы на небольшие подсистемы, каждая из которых имеет небольшой набор функций пригодности.

Например, некоторые компании имеют дело с данными, требующими особых мер обеспечения безопасности, такими как номер кредитных карт или детали платежей. В зависимости от отрасли или характера работы, хранение информации такого типа предполагает выполнение серьезных юридических требований, которые могут меняться из-за изменений законодательства или стандартов, влияющих на нормы в связи с охватом новых территорий или стран, имеющих другие правовые требования.

Если разработчики устанавливают, что безопасность и стоимость системы играют значительную роль для функции пригодности всей системы, это может привести к необходимости проектирования архитектуры, которая рассматривает два этих фактора вместе. Не подобрав функции пригодности на ранней стадии проектирования, команда может столкнуться с тем, что эти факторы окажутся разбросаны по всей базе программного обеспечения, требуя более широкого анализа

их влияния, чтобы понять изменения и увеличение общей стоимости затрат на модификацию.

Функции пригодности можно разделить на три простые категории:

Ключевые

Это области, которые важны для выбора технологии или метода проектирования. Необходимо приложить больше усилий для изучения вариантов дизайна, которые значительно облегчают изменение этих элементов. Например, для банковских приложений важны производительность и отказоустойчивость.

Релевантные

Эти области необходимо учитывать на объектном уровне, но они не могут определять выбор архитектуры. Например, метрики качества базы кода важны, но не являются ключевыми.

Не релевантные

На выбор проекта и технологии эти области влияния не оказывают. Например, метрики процесса, такие как время цикла (необходимое время для перехода от проектирования к реализации), могут иметь некоторое значение, но не для архитектуры. Поэтому функция пригодности для этих областей не нужна.



Следите за показаниями ключевых и релевантных функций пригодности, выкладывая результаты выполнения на видном месте или в общем пространстве, чтобы разработчики могли их учитывать в повседневной работе с кодом.

Классификация функций пригодности по категориям помогает определить приоритетные проектные решения. Если проект решения имеет определенные последствия для ключевой функции пригодности, для проверки архитектурных аспектов дизайна стоит потратить больше времени и усилий на проведение спайков (spikes, ограниченных во вре-

мени проектов экспериментального программирования). Некоторые команды используют комплексную разработку¹, практику бережливых и гибких процессов для параллельного проектирования нескольких решений, оставляя возможные варианты открытыми для принятия будущих решений в обмен на затраты, необходимые для построения множества решений.

Пересмотр функций пригодности

Пересмотр функций пригодности представляет собой встречу с ключевыми заинтересованными сторонами с целью обновления функций пригодности для выполнения целей проектирования. Такие события, как значительный рост рынка или числа заказчиков, появление новой области функционирования или бизнес-возможностей либо реконструкция существующей части системы, могут гарантировать пересмотр функции пригодности.

Пересмотр функции пригодности обычно предусматривает следующее:

- Анализ существующих функций пригодности.
- Проверку актуальности существующих функций пригодности.
- Определение изменения масштаба или размера каждой функции пригодности.
- Решение относительно использования более подходящих методов измерения или тестов функций пригодности.
- Выявление новых функций пригодности, которые система должна поддерживать.



Пересматривайте ваши функции пригодности не реже одного раза в год.

¹ https://ru.wikipedia.org/wiki/Гибкая_методология_разработки

ЭЛЕКТРОННАЯ ТАБЛИЦА PENULTIMATEWIDGETS И АРХИТЕКТУРЫ ПРЕДПРИЯТИЯ

Когда архитекторы приложения PenultimateWidgets решили построить новую платформу для проекта, сначала они создали электронную таблицу со всеми требуемыми характеристиками: масштабируемости, безопасности, отказоустойчивости и множества других возможностей. Но они столкнулись с известной проблемой: если строить новую архитектуру для поддержки всех этих характеристик, как можно гарантировать, что она сохранит эту поддержку? По мере добавления новых характеристик как можно будет сохранить эти важные характеристики от неожиданного ухудшения?

Решением было создать новые функции пригодности для каждой из характеристик в электронной таблице, переформулировав некоторые из них для выполнения критерия объективной оценки. В отличие от случайной, специальная проверка критерия их важности включает функции пригодности в конвейер развертки (более подробное описание см. в главе 3).

В то время как разработчики архитектуры заинтересованы в изучении архитектур с эволюционным развитием, мы не делаем попыток моделировать биологическую эволюцию. Теоретически можно построить архитектуру, которая случайным образом меняет один из своих битов (мутирует) и заново развертывает себя. Через несколько миллионов лет мы, вероятно, получим очень интересную архитектуру. Но мы не можем столько ждать.

Мы хотим, чтобы наша архитектура эволюционировала управляемым образом, поэтому накладываем ограничения на различные аспекты архитектуры для исключения нежелательных направлений эволюционирования. Хорошим примером является собаководство: выбирая желаемые характеристики, мы можем создать широкий ряд различных пород собак за относительно короткий промежуток времени.

В следующей главе мы рассмотрим больше аспектов использования функций пригодности. В главе 6 мы объединяем функции пригодности со всеми остальными областями архитектуры.

3

Проектирование инкрементных изменений

Архитектура с эволюционным развитием поддерживает управляемые, *инкрементные* изменения в нескольких областях.

— *Наше определение*

В 2010 году Джез Хамбл (Jez Humble) и Дэйв Фарлей (Dave Farley) выпустили *Систему непрерывной поставки* (<https://continuousdelivery.com/>), сборник практических советов по повышению эффективности проектирования программного обеспечения. Они разработали *механизм* построения и релизов программного обеспечения с помощью автоматизации и специальных инструментов, но не *структуры*, позволяющей проектировать программное обеспечение с возможностью эволюционного развития. Эволюционирующая архитектура предполагает использовать эту практику проектирования в качестве предварительных условий проектирования эволюционирующего программного обеспечения.

Наше определение эволюционирующей архитектуры предполагает *инкрементное изменение*, означающее то, что архитектура должна способствовать изменению небольшими приращениями. В этой главе описывается архитектура, которая поддерживает постепенные изменения наряду с использованием некоторой практики проектирования для достижения постепенных изменений, что является важным элементом строительства эволюционирующей архитектуры.

Мы обсудим два этапа процесса реализации постепенных изменений: *разработку*, которая включает в себя то, как разработчики создают программное обеспечение, и *ввод в эксплуатацию*, который включает в себя то, как команды развертывают программное обеспечение.

Далее приводится пример эксплуатационного этапа процесса инкрементных изменений. Начнем с примера инкрементных изменений из главы 1, который включает дополнительные детали архитектуры и среды развертывания. У приложения PenultimateWidgets, нашего поставщика виджетов, есть страничка с каталогом, опирающаяся на архитектуру микросервисов и опыт проектирования (рис. 3.1).

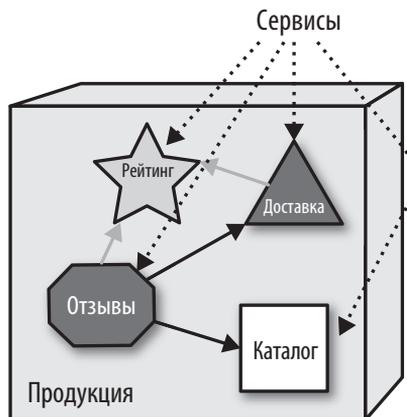


Рис. 3.1. Первоначальная конфигурация развертывания компонентов приложения PenultimateWidgets

Разработчики PenultimateWidgets использовали микросервисы, которые изолированы от остальных сервисов. Микросервисы используют архитектуру *без разделения ресурсов*: каждый сервис в своей работе не связан с другими, чтобы исключить техническую связанность и тем самым способствовать изменению на детальном уровне. Приложение PenultimateWidgets развертывает все свои сервисы в отдельных контейнерах для упрощения изменений режимов работы.

Этот веб-сайт позволяет пользователям оценить различные виджеты по звездочной системе оценок. Но остальные части архитектуры

также нуждаются в оценке (представителями клиентских сервисов, поставщиками транспортных услуг и т. п.), так что все они совместно используют звездочную систему оценок. Однажды команда, занимающаяся оценкой, выпускает новую версию наряду с существующей, что позволяет использовать промежуточные категории, являющиеся значимым обновлением (рис. 3.2).



Рис. 3.2. Развертывание с использованием сервиса улучшенной оценки, показывающее промежуточные оценочные категории

Сервисы, которые используют рейтинги, не требуют единовременного перехода на сервис улучшенных оценок, а могут постепенно переходить на улучшенный сервис, когда это будет удобно. Со временем все большее число частей экосистемы нуждается в переходе на усиленную версию системы оценок. Часть практики DevOps в PenultimateWidgets включает в себя архитектурный мониторинг не только сервисов, но также маршрутов между ними. Когда группа контроля операций видит, что ни одна из операций не направляется в определенный сервис в течение заданного интервала времени, группа автоматически выводит этот сервис из экосистемы, как показано на рис. 3.3.

Механическая способность к эволюции является одним из компонентов эволюционирующей архитектуры. Рассмотрим приведенную выше абстракцию, опустившись на один уровень глубже.

Приложение PenultimateWidgets имеет мелкоструктурную архитектуру микросервисов, в которой каждый сервис развернут с помощью контейнера (такого, как Docker, <https://www.docker.com/>) и паттерна



Рис. 3.3. Все сервисы теперь используют улучшенный сервис оценки

сервиса для обработки микроструктурных связей. Используемые `PenultimateWidgets` приложения состоят из маршрутов между выполняемыми экземплярами сервисов, при этом сервис может иметь несколько экземпляров для обработки возникающих при работе задач, таких как выполняемое по требованию масштабирование. Это позволяет разработчикам принимать в работу различные версии сервисов и с помощью маршрутизации контролировать доступ. Когда конвейер разворачивает тот или иной сервис, он регистрирует себя (положение и контракт) с помощью инструмента обнаружения сервиса. Когда одному сервису требуется найти другой, они используют инструмент обнаружения, чтобы узнать пригодность версии и соответствующий контракт.

При разворачивании нового сервиса звездочного рейтинга он регистрирует сам себя с помощью инструмента обнаружения и публикует новый контракт. Новая версия сервиса поддерживает более широкий диапазон значений (а именно, половинчатые значения), чем исходный. Это значит, что разработчикам сервиса можно не беспокоиться об ограничении поддерживаемых значений. Если в новой версии для вызывающих программ требуется другой контракт, это обычно решается самим сервисом, а не вызывающими кодами, решающими, какую версию вызвать. Мы рассматриваем эту стратегию контракта в «Версиях внутренних сервисов» на с. 195.

Когда команда развертывает новый сервис, она не хочет форсировать немедленное обновление вызовов сервисов на вызов новых версий. Поэтому разработчик временно меняет конечную точку сервисов звездочной оценки рейтинга в модуле доступа, который проверяет, какая версия сервиса требуется, и направляет в требуемую версию. Никакие существующие сервисы не должны изменяться, чтобы пользоваться сервисом оценки рейтинга как всегда, но новые вызовы могут начать пользоваться преимуществами новых возможностей. Старые сервисы не обновляются и могут продолжать вызывать исходный сервис так долго, как требуется. По мере того как сервисы вызова принимают решение использовать новое поведение, они меняют версию, которую они запрашивают из конечной точки. Со временем первоначальной версией прекращают пользоваться, и в какой-то момент разработчик архитектуры может удалить старую версию из конечной точки, когда она больше не нужна. В ходе выполнения операций выявляются сервисы, которые больше не вызываются (в пределах разумного порогового значения), и выполняется их утилизация.

Все изменения этой архитектуры, включая инициализацию внешних компонентов, таких как базы данных, выполняются под контролем конвейера развертывания, снимающего с DevOps ответственность за координацию различных перемещений при развертывании отдельных частей.

В этой главе описываются характеристики, опыт проектирования, соображения команд и другие аспекты построения архитектуры, которые поддерживают постепенные изменения.

Строительные блоки

Многие строительные блоки, необходимые для гибкости архитектуры, получили за последние несколько лет широкое применение при поддержке системы непрерывной поставки.

Разработчики должны определить, в какой степени их системы соответствуют друг другу, создавая для этого соответствующие схемы. Архитекторы часто попадают в ловушку, рассматривая архитектуру программного обеспечения как *уравнение*, которое они должны решить. Большая часть коммерческого оборудования, проданного разработчикам архитектуры, усиливает математическую иллюзию определенности благодаря блокам, линиям и стрелкам на этих схемах. Несмотря на определенную пользу эти схемы представляют собой двумерные виды или копии изображений идеального мира, но мы все живем в четырехмерном мире. Для детализации этих двумерных схем необходимо добавить конкретные характеристики. Маркировка объектно-реляционного отображения (ORM) на рис. 3.4 становится JDBC 2.1, что обеспечивает трехмерное отображение мира, в котором разработчики проверяют свои проекты в реальной производственной среде, используя реальное для этого программное обеспечение. Как показано на рис. 3.4, происходящие со временем изменения в предметной и технологической областях требуют от разработчика адаптировать четырехмерный вид архитектуры, превращая эволюцию в функциональность первого класса.

В программном обеспечении ничто не является статичным. Возьмем, например, компьютер. Установите на нем операционную систему и набор программного обеспечения, затем закройте компьютер в кабинете на год. В конце года достаньте его из кабинета, включите питание и интернет... и очень долго вам придется наблюдать, как на нем устанавливаются обновления. Даже если на компьютере не был изменен ни один бит, *весь окружающий мир продолжал изменяться*; это является динамическим равновесием, которое мы уже описывали. Любое разумное планирование архитектуры должно включать эволюционные изменения.

Когда мы знаем, как ввести архитектуру в производство и обновить ее, чтобы включить неизбежные изменения (обновления системы безопасности, новые версии программного обеспечения, постепенные изменения архитектуры и т. п.), мы продвигаемся в четырехмерный

мир. Архитектура не является неизменным уравнением, она представляет собой срез текущего состояния определенного процесса, как это показано на рис. 3.4.

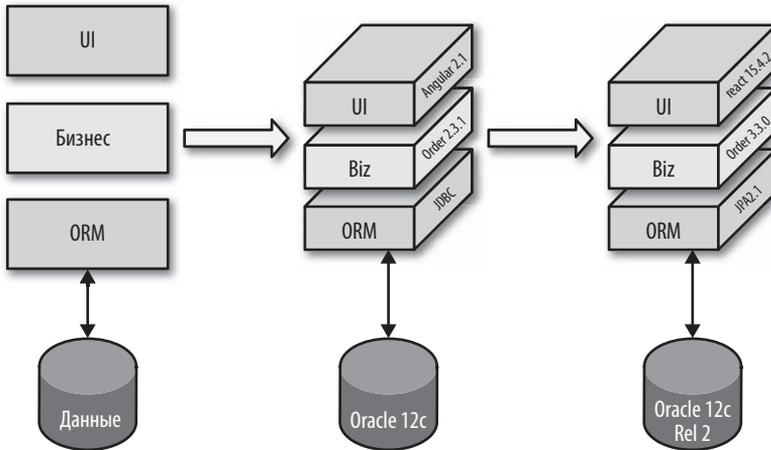


Рис. 3.4. Современная архитектура должна иметь возможность развертываться и меняться, чтобы выжить в реальном мире

Система непрерывного развертывания и перемещения DevOps подтверждают необходимость реализации архитектуры и поддержания ее функциональности. Нет ничего ошибочного в использовании модели архитектуры и в направленных на это усилиях, но создание модели является только первым шагом.



Архитектура является абстрактным понятием до тех пор, пока не введена в эксплуатацию, после этого она становится живым существом.

На рис. 3.4 представлена естественная эволюция версии в процессе ее обновлений и выбора новых инструментов. Архитектуры эволюционируют также и по-другому, как будет описано в главе 6.

Разработчики не могут судить о долговременной жизнеспособности любой архитектуры, пока не пройдут успешно этапы *проектирования, реализации, обновления и неизбежных изменений*. И, вероятно, даже позволяли архитектуре выдерживать необычные происшествия, основываясь на появляющихся неизвестных, которые мы описываем в главе 6.

Тестопригодность

Одной из часто игнорируемых возможностей архитектуры программного обеспечения является *тестопригодность*, а именно способность характеристик архитектуры проходить автоматизированные тесты. К сожалению, часто бывает сложно проверить определенные части архитектуры из-за отсутствия инструментальной поддержки.

Однако некоторые аспекты архитектуры допускают несложное тестирование. Например, разработчики могут проверять конкретные характеристики архитектуры, такие как связанность, разрабатывать инструкции и в конечном счете автоматизировать эти тесты.

Далее приводится пример функции пригодности, определенной в области технической архитектуры для контроля направленности связи между компонентами. В экосистеме Java JDepend является инструментом, который анализирует характеристики связи пакетов. Поскольку инструмент JDepend написан на Java, он имеет интерфейс приложений, который разработчики могут применить для проведения собственного анализа с помощью модульных тестов.

Рассмотрим функцию пригодности из примера 3.1, выраженную как проверка JUnit.

Пример 3.1. 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 3.1 мы определили пакеты нашего приложения, а затем задали правила импорта. Одной из проблем, возникающих в основанной на компонентах системе, являются циклы компонентов, то есть когда компонент А обращается к компоненту В, который затем снова обращается к компоненту А. Если разработчик случайно написал код, который импортирует данные в `util` из `persistence`, модульное тестирование провалится до завершения работы кода. Мы предпочитаем писать модульные тесты для обнаружения нарушений в архитектуре при использовании строгих инструкций разработки (со всеми бюрократическими проволочками): это позволяет разработчикам больше сосредоточиться на проблеме рассматриваемой области и меньше на проблемах настройки. Что еще более важно, это позволяет разработчикам объединять правила в виде выполняемых артефактов.

Функции пригодности могут иметь любого владельца, включая общего. В примере 3.1 команда может владеть функцией пригодности направленности, потому что она является определенной функциональностью проекта. В том же самом конвейере развертывания общей для многих проектов функцией пригодности может владеть команда обеспечения безопасности. В общем случае ответственность за определение и поддержание функций пригодности распределяется между архитекторами и любыми другими специалистами, занимающимися поддержанием целостности архитектуры.

Многие характеристики архитектуры тестопригодные. Существуют инструменты для проверки структурных характеристик архитектуры, такие как JDepend (или аналогичный инструмент NDepend в экосистеме .NET, <https://www.ndepend.com/>). Есть соответствующие

инструменты для проверки производительности, масштабируемости, отказоустойчивости и других характеристик архитектуры. Доступны также инструменты мониторинга и регистрации: любой инструмент, который помогает оценить некоторые характеристики архитектуры, расценивается как функция пригодности.

После того как были определены функции пригодности, разработчики должны убедиться, что они производят оценку своевременно. Автоматизация является ключом к непрерывной проверке. *Конвейер развертывания* (deployment pipeline) часто используется для оценки задач, таких как эта. Используя конвейер развертывания, разработчики могут определять чем, когда и как часто выполняются функции пригодности.

Конвейеры развертывания

Непрерывная поставка (continuous delivery) описывает механизм конвейера развертывания. Аналогично серверу непрерывной интеграции, конвейер развертывания «прислушивается» к изменениям, затем выполняет серию проверочных шагов, каждый с более высокой сложностью. Практика непрерывной поставки способствует использованию конвейера развертывания в качестве механизма автоматизации общих задач проекта, таких как проверки, инициализация машины, развертывания и т. п. Имеющиеся в свободном доступе инструменты, такие как GoCD (<https://www.gocd.org/>), способствуют построению этих конвейеров развертывания.

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ И КОНВЕЙЕР РАЗВЕРТЫВАНИЯ

Непрерывная интеграция — это хорошо известная инженерная практика в гибких проектах, которая поощряет разработчиков выполнять интеграцию как можно раньше и как можно чаще. Для облегчения непрерывной интеграции появились инструменты ThoughtWorks под названием CruiseControl (<http://cruisecontrol.sourceforge.net/>), а также коммерческие предложения с открытым исходным кодом. Непрерывная интеграция предоставляет «официальное» место сборки, и разработчики пользуются концепцией одного механизма для обес-

печения работоспособного кода. Однако сервер непрерывной интеграции также предоставляет точное время и место для выполнения общих задач проекта, таких как модульное тестирование, зона охвата проверкой кода, метрические показатели, функциональные проверки и т. п. Для многих проектов сервер непрерывной интеграции включает список заданий, чье успешное завершение указывает на надлежащее выполнение разработки. Крупные проекты в конечном счете составляют впечатляющий список заданий.

Конвейеры развертывания помогают разработчикам разбивать отдельные задачи на *этапы*. Конвейер развертывания включает концепцию поэтапного построения, позволяющую разработчикам моделировать столько заданий после проверки, сколько будет необходимо. Эта возможность разделять задание на отдельные этапы поддерживает более широкие полномочия конвейера развертывания, а именно проверять готовность к производству, в отличие от сервера непрерывной интеграции (CI — continuous integration), который в первую очередь акцентируется на интеграции. Поэтому конвейер развертывания обычно включает в себя проверку приложений на нескольких уровнях, автоматическую инициализацию среды и много других проверок.

Некоторые разработчики пытаются «обойтись» сервером непрерывной интеграции, но вскоре обнаруживают, что им не хватает уровня разделения заданий обратной связи.

Типичный конвейер развертывания автоматически создает среду развертывания (похожую на контейнер Docker или специальную среду, создаваемую таким инструментом, как Puppet (<https://puppet.com/>) или Chef (<https://www.chef.io/chef/>)), как показано на рис. 3.5.

Образ развертывания, выполняемый конвейером, позволяет разработчикам и группам эксплуатации обеспечить высокий уровень достоверности: главный компьютер (или виртуальная машина) определяется декларативно, и обычно на практике его создают заново из ничего.

Конвейер развертывания также предлагает идеальный способ выполнения функций пригодности, определенных для архитектуры: оказывается, критерий произвольной проверки имеет несколько

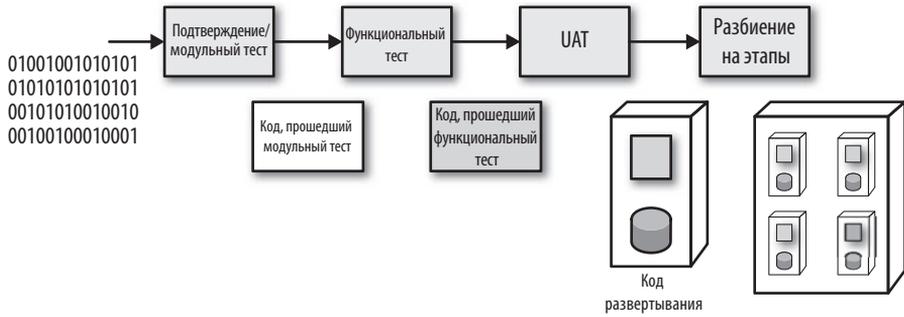


Рис. 3.5. Этапы развертывания конвейера

этапов для включения различных уровней абстракции и сложности проверок и выполняется всякий раз при любом изменении системы. Конвейер развертывания с добавленными функциями пригодности представлен на рис. 3.6.

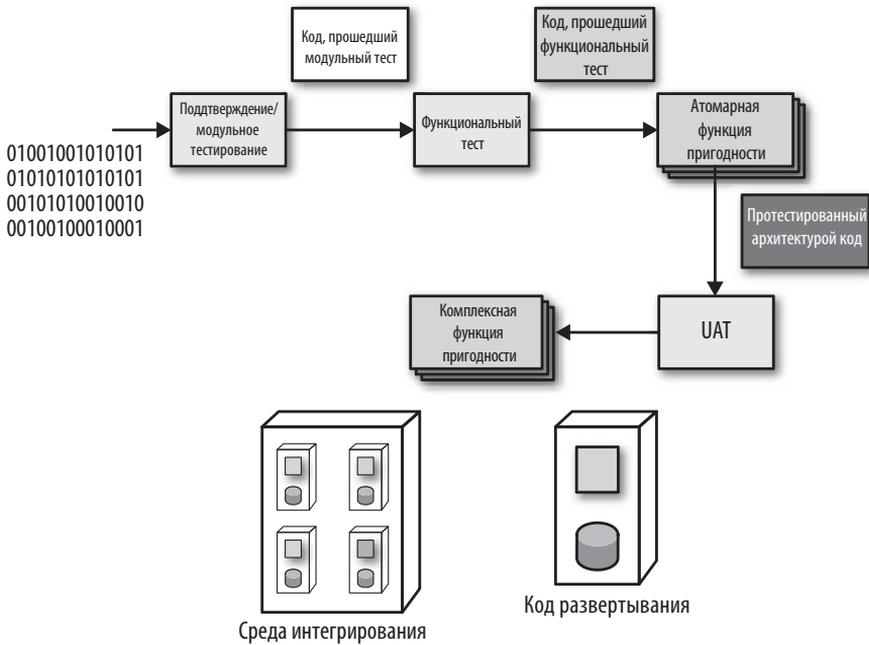


Рис. 3.6. Конвейер развертывания с функциями пригодности, добавленными в виде этапов

На рис. 3.6 показан набор атомарных и комплексных функций пригодности (последние используются в более сложной среде с интеграцией). Конвейеры развертывания могут обеспечить правила защиты архитектурных областей, которые выполняются всякий раз при изменении системы.

КОНВЕЙЕРЫ РАЗВЕРТЫВАНИЯ `PenultimateWidgets`

В главе 2 мы описали электронную таблицу с требованиями `PenultimateWidgets`. После того как была использована практика проектирования непрерывной поставки, стало понятно, что нефункциональные требования платформы работают лучше в автоматизированном конвейере развертывания. В связи с этим разработчики сервисов создали конвейер развертывания для проверки функций пригодности, созданных разработчиками корпоративных архитектур и командой этого сервиса. Теперь всякий раз, когда команда вносит изменение в сервис, барьер проверок выполняет проверки кода и пригодности архитектуры в целом.

Другой распространенной практикой, используемой в проектах с эволюционирующей архитектурой, является непрерывное развертывание, использующее конвейер развертывания, чтобы внести изменения в состав операций ввода в эксплуатацию при успешном прохождении конвейера тестов. Несмотря на то что непрерывное развертывание является идеальным методом, оно требует сложную координацию: разработчики должны обеспечить, чтобы изменения, развернутые для ввода в эксплуатацию на постоянной основе, не нарушали порядок системы.

Для решения проблемы координации обычно используют в конвейерах развертывания операцию *ветвления* (*fan out*), с помощью которой конвейеры параллельно выполняют несколько заданий, как это показано на рис. 3.7.

Как показано на рис. 3.7, когда команда делает изменение, им необходимо проверить две вещи: то, что при этом нет отрицательного влияния на текущее состояние производства (потому что успешное

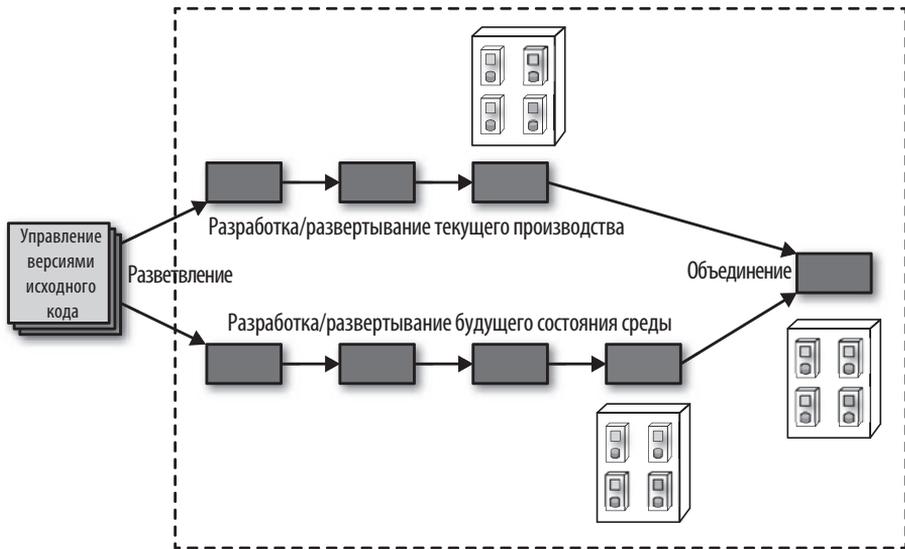


Рис. 3.7. Ветвление в конвейере развертывания для проверки нескольких сценариев

выполнение развертывания конвейером позволит ввести код в эксплуатацию) и что их изменения были успешными (влияющими на будущее состояние среды). Ветвление конвейера развертывания дает возможность параллельно выполнять несколько заданий (тестирования, развертывания и т. п.), экономя при этом время. После того как серия параллельно выполняемых заданий, представленных на рис. 3.7, завершится, конвейер может оценить результаты, и если все задания завершены успешно, выполнить операцию *объединения* (*fan in*), собрав в один поток все действия для выполнения развертывания. Обратите внимание на то, что конвейер развертывания может выполнить эту комбинацию *ветвления* и *объединения* много раз всякий раз, когда команде требуется оценить изменения в нескольких контекстах.

Другой часто встречающейся проблемой, связанной с конвейером развертывания, является его воздействие на предметную область. Пользователи не хотят, чтобы новые функции отображались на регулярной основе, но предпочли бы, чтобы они были организованы более традиционным способом, например развертыванием «Big Bang».

Распространенный способ непрерывного развертывания вместе с поэтапными обновлениями состоит в использовании *ротации функций* (feature toggles). Внедряя новые функции, скрытые под функциями ротации, разработчики могут безопасно развернуть эту функцию для введения в эксплуатацию, не беспокоясь о том, что пользователи увидят ее преждевременно.

ОБЕСПЕЧЕНИЕ КАЧЕСТВА В ПРОЦЕССЕ ПРОИЗВОДСТВА

Одним из полезных побочных эффектов постоянного создания новых функций с использованием ротации функций является возможность выполнять в процессе ввода в эксплуатацию задачи по контролю качества. Многие компании не понимают, что они могут использовать свою производственную среду для исследовательского тестирования. После того как команде стало удобно использовать ротацию функций, она может развернуть эти изменения в производстве, поскольку большинство платформ ротации функций позволяют разработчикам прокладывать маршруты пользователям на основе широкого разнообразия критериев (IP-адресов, списка управления доступом (ACL — access control list) и т. п.). Если команда разворачивает новые функции внутри области ротации функций, к которой имеет доступ только отдел контроля качества, то они могут проводить проверки в процессе ввода в эксплуатацию.

Применение при проектировании конвейера развертывания дает возможность разработчикам архитектуры без труда использовать функции пригодности проекта. Необходимость оценки того, какие этапы нужны при разработке, является распространенной проблемой, возникающей при проектировании конвейера. Преобразование функциональностей архитектуры проекта (включая сохранение возможности эволюционного развития) в функции пригодности дает ряд преимуществ:

- Функции пригодности предназначены давать объективные, количественные результаты.
- Использование всех функциональностей в качестве функций пригодности создает механизм согласованного усиления.

- Составление списка функций пригодности позволяет самым простым образом проектировать конвейер развертывания.

Определение того, когда и какую функцию пригодности запускать в цикле сборки проекта, а также использование надлежащего контекста является нетривиальной задачей. Однако после того, как функции пригодности внутри конвейера развертывания оказались на своих местах, разработчики получают высокий уровень доверия в отношении того, что эволюционные изменения не нарушат требований к проекту. Функциональные возможности архитектуры часто недостаточно хорошо поясняются и редко, при этом часто субъективно, оцениваются; создание на их основе функции пригодности позволит более строго и поэтому с большим уровнем доверия использовать их в практике проектирования.

Комбинирование категорий функций пригодности

Категории функций пригодности часто пересекаются при их использовании в таких механизмах, как конвейеры развертывания. Здесь приводятся некоторые распространенные комбинации категорий функций пригодности вместе с соответствующими примерами.

атомарные + триггерные

Примером этого типа функции пригодности являются модульные и функциональные тесты, выполняемые как часть процесса разработки программного обеспечения. Разработчики выполняют их для проверки изменений и механизма автоматизации, такого как конвейер развертывания, выполняя непрерывную интеграцию для обеспечения своевременности. Общеизвестным примером этого типа функции пригодности является модульное тестирование, которое проверяет некоторые аспекты целостности архитектуры, такие как круговые зависимости или цикломатическая сложность.

комплексные + триггерные

Комплексные триггерные функции пригодности являются частью проверки целостности с помощью конвейера развер-

тивания. Разработчики проектируют эти тесты специально для проверки того, как взаимодействуют между собой различные аспекты системы, для которой предусмотрены вполне определенные типы взаимодействия. Например, разработчики могут быть заинтересованы в том, чтобы знать, какой тип взаимодействия осуществляет подключаемая система безопасности на масштабируемость. Архитекторы проектируют эти тесты для преднамеренного тестирования некоторой интеграционной характеристики в базе кода, поскольку поломки указывают на недостаток архитектуры. Подобно всем триггерным тестам, разработчики обычно запускают эти функции пригодности во время разработки и в качестве части конвейера развертывания или среды непрерывной интеграции. В общем случае это относится к тестам и метрикам, которые имеют хорошо известные выходные данные.

атомарные + непрерывные

Непрерывные тесты выполняются как часть разработки архитектуры, и разработчики проектируют системы с учетом их присутствия. Например, разработчиков архитектуры может беспокоить, что все инструменты проверок конечных точек в службе REST, поддерживающие соответствующие операторы обработки данных, указывают на ошибку обработки, и при этом поддерживают надлежащим образом метаданные и поэтому они создают инструмент, чтобы непрерывно вызывать конечные точки REST (то, что будет делать клиент) и проверять результаты. *Атомарность* этих функций пригодности предполагает, что они тестируют только один аспект архитектуры, а *непрерывность* указывает, что тесты выполняются как часть всей системы.

комплексные + непрерывные

Комплексная, непрерывная функция пригодности все время проверяет многочисленные части системы. В основном этот механизм представляет агента (или еще одного клиента) в системе, которая непрерывно оценивает комбинацию архитектур-

ных и рабочих качеств. Превосходным примером непрерывной комплексной функции пригодности в реальном мире является программа Chaos Monkey компании Netflix (<https://github.com/netflix/chaosmonkey>). Когда Netflix разработала дистрибутив своей архитектуры, они спроектировали его для работы в Amazon Cloud. Но инженеров беспокоило, какой тип необычного поведения может проявиться, ведь у них не было непосредственного контроля над своими разработками, а именно: над появлением большого времени задержки, изменением пригодности, гибкости системы в облачной среде. С этой целью была создана Chaos Monkey, за которой последовала программа Simian Army (<https://github.com/Netflix/SimianArmy>). Функция Chaos Monkey «просачивается» в самый центр данных Amazon и запускает череду неожиданных событий: время задержки растет, надежность снижается и наступает хаос. С учетом Chaos Monkey каждая команда должна создать отказоустойчивые сервисы. Инструмент проверки RESTful, упомянутый в предыдущем разделе, существует в виде Conformity Monkey (<https://github.com/Netflix/SimianArmy/wiki/Conformity-Home>), который проверяет каждый сервис для выбора наилучшей практики проектирования в зависимости от архитектуры.

Обратите внимание, что Chaos Monkey не является инструментом проверок, запускаемым по графику, так как он работает непрерывно в рамках экосистемы Netflix. Это позволяет создавать системы, способные преодолевать проблемы. Этот инструмент также непрерывно проверяет пригодность системы. Использование такой постоянной проверки, встроенной в архитектуру, дало возможность Netflix создать одну из наиболее отказоустойчивых систем в мире. Программа Simian Army является превосходным примером комплексной непрерывно работающей операционной функции пригодности. Она выполняется на многочисленных частях архитектуры сразу, обеспечивая сохранение характеристик архитектуры (таких, как отказоустойчивость, масштабируемость).

Комплексные, непрерывно работающие функции пригодности относятся к наиболее сложным функциям пригодности по реализации,

но могут обеспечить высокую продуктивность, как это показано в следующем примере.

Практический пример: реструктуризация архитектуры при ее развертывании 60 раз в день

GitHub является хорошо известным сайтом разработчиков с агрессивной практикой проектирования, развертываемым в среднем 60 раз в день. Они описывают эту проблему в своем блоге «Шевелись быстрее и решай проблемы» (<http://githubengineering.com/move-fast/>), что заставляет вздрагивать от ужаса многих разработчиков архитектуры. Оказалось, что GitHub длительное время использовал сценарий оболочки, обернутой вокруг командной строки Git для обработки слияний, которые хотя и работают правильно, но недостаточно хорошо масштабируются. Команда инженеров Git построила библиотеку замены многих командных строк функций Git, которая получила название libgit2. Там использовалась функция слияния, предварительно тщательно протестированная на месте.

Но теперь они должны внедрить новое решение в производство. Такое поведение было частью GitHub с момента его создания, и это работало безупречно. Последнее, что хотели бы сделать разработчики, это добавить ошибки в существующую функциональность, но они также должны были вернуть технический долг.

К счастью, разработчики GitHub создали общедоступную платформу Scientist (<https://github.com/github/scientist>), которая предоставляет комплексную, непрерывно проверяющую функцию для тестирования изменений в коде. В примере 3.2 приводится структура теста Scientist.

Пример 3.2. Настройка Scientist для эксперимента

```
require "scientist"

class MyWidget
  include Scientist

  def allows?(user)
    science "widget-permissions" do |e|
```

```
    e.use { model.check_user(user).valid? } # старый способ
    e.try { user.can?(:read, model) } # новый способ
  end # returns the control value
end
end
```

В примере 3.2 разработчики использовали имеющиеся характеристики блока `use` (назовем его *контрольным*) и добавили экспериментальные характеристики в блок `try` (назовем его *испытываемым*). Блок `Scientist` при вызове кода обрабатывает следующие данные:

Решает, запускать или нет блок try

Разработчики настроили блок `Scientist` на определение того, как выполняется эксперимент. Например, в рассматриваемом случае, цель которого — обновить объединенную функциональность, 1 % случайно выбранных пользователей пытались использовать новую объединенную функциональность. В любом случае, `Scientist` *всегда* возвращает результаты блока `use`, гарантируя, что при наличии различий пользователь всегда получал существующую функциональность.

Рандомизирует порядок выполнения блоков use и try

`Scientist` использует случайный порядок выполнения, чтобы предотвратить возможную маскировку ошибок из-за неизвестных зависимостей. Иногда порядок выполнения или другие случайные факторы могут вызвать ложные срабатывания; при случайном порядке этот инструмент делает такие срабатывания маловероятными.

Измеряет продолжительность всех типов поведения

Часть работы `Scientist` состоит в А/В-тестировании производительности, поэтому в блок был встроен контроль производительности. Фактически, разработчики могут использовать фреймворк по частям, например, они могут считать вызовы, не проводя экспериментов.

Сравнивает результат try с результатом use

Поскольку целью является перенастройка существующего поведения, `Scientist` сравнивает и регистрирует результаты каждого вызова, чтобы обнаружить, существует ли различие.

Принимает как есть (но регистрирует) любые исключительные ситуации, возникающие в блоке try

Всегда есть вероятность того, что новый код выдаст неожиданные исключения. Разработчики не хотят, чтобы конечные пользователи видели эти ошибки, поэтому инструмент делал их невидимыми для конечных пользователей (при этом все исключительные ситуации регистрировались, чтобы разработчик смог их проанализировать).

Публикует всю информацию

`Scientist` обеспечивает доступность всех данных в различных форматах.

Для объединения рефакторинга разработчики GitHub использовали процедуру тестирования новой реализации системы (названной `create_merge_commit_rugged`), как это показано в примере 3.3.

Пример 3.3. Экспериментирование с новым алгоритмом 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
```

В примере 3.3 вызов `create_merge_commit_rugged` выполнялся в 1 % всех вызовов, но, как отмечалось в рассматриваемом случае, в масштабе GitHub все пограничные случаи появлялись быстро.

Если исполняется код, конечные пользователи всегда получают правильные результаты. Когда же блок `try` возвращает значение, отличное от блока `use`, этот случай регистрируется и возвращается значение блока `use`. Поэтому, в худшем случае, конечный пользователь получает в точности то значение, которое он получил бы до рефакторинга. После выполнения эксперимента в течение 4 дней, если в течение 24 часов не наблюдается никаких случаев задержки работы или несовпадающих результатов, разработчики удаляют старый код объединения и оставляют вместо него новый.

С нашей точки зрения, `Scientist` является функцией пригодности. Рассмотренный случай — это яркий пример стратегического использования комплексной, непрерывно работающей функции пригодности, которая может дать возможность разработчикам с уверенностью произвести рефакторинг критически важной части инфраструктуры их системы. Разработчики изменили ключевую часть архитектуры, запустив новую версию с существующей, особенно тщательно преобразуя использование старой версии при проверке на совместимость.

В общем случае, архитектура будет использовать большое число атомарных функций пригодности и несколько комплексных функций. Определяющий фактор атомарности сводится к тому, что разработчики проверяют также, насколько всеобъемлющими являются полученные результаты.

Конфликтующие цели

Процесс разработки гибкого программного обеспечения научил нас, что чем раньше будет обнаружена проблема, тем меньше усилий потребуются для ее разрешения. Одним из побочных эффектов совместного учета всех областей архитектуры программного обеспечения является выявление на ранней стадии конфликта целей

разных областей. Например, в организации могут выразить желание использовать наиболее агрессивный такт изменений, чтобы поддерживать новые функции. Быстрое изменение кода предполагает быстрые изменения схем организации баз данных, но администраторы баз данных в большей степени обеспокоены сохранением стабильности, потому что они строят хранилище данных. Две цели эволюции конфликтуют между технической частью архитектуры и архитектурой баз данных.

Очевидно, что необходим некоторый компромисс, учитывающий множество факторов, влияющих на лежащую в основе архитектуры объектную область. Использование всех областей архитектуры как способа выявления проблемных частей (с дополнительным использованием функций пригодности для их оценки) дает возможность сравнивать их между собой, предоставляя больше информации для расстановки приоритетов.

Конфликт целей неизбежен. Однако обнаружение и количественное определение этих конфликтов на раннем этапе позволяет архитекторам принимать более обоснованные решения и создавать более четкие цели и принципы.

Практический пример: добавление функций пригодности в сервис выставления счетов PenultimateWidgets

Приведенная нами в качестве примера компания PenultimateWidgets использует архитектуру, содержащую сервис подготовки счетов. Команда выставления счетов хотела бы обновить библиотеки и используемые методы, чтобы при этом изменения не влияли на возможность других команд интегрироваться с ними.

Команда выставления счетов определила следующие требования:

Масштабируемость

Производительность не очень важна для PenultimateWidgets, так как они обрабатывают детали счетов-фактур для нескольких

поставщиков, поэтому сервис выставления счетов должен поддерживать доступность соглашения об уровне обслуживания.

Интеграция с другими сервисами

Несколько других сервисов в экосистеме PenultimateWidgets используют выставление счетов. Команда хотела бы убедиться в том, что точки интеграции сервисов не пропадут при внутренних изменениях.

Безопасность

Выставление счетов связано с движением денежных средств, и безопасность в этих случаях всегда вызывает беспокойство.

Контролируемость

Некоторые государственные нормативы требуют, чтобы изменения кода в программе налогообложения проверялись независимым контролером.

Команда выставления счетов использует сервер непрерывной интеграции, который недавно был обновлен до выполнения по требованию инициализации среды, в которой выполняется код. Для использования функций пригодности эволюционирующей архитектуры они вместо сервера непрерывной интеграции включили в систему конвейер развертывания, что позволило создать несколько этапов выполнения, как показано на рис. 3.8.

Конвейер развертывания PenultimateWidgets включает в себя шесть этапов.

1 этап — репликация непрерывной интеграции (CI)

На этом этапе происходит репликация поведения бывшего сервера CI, выполнение модульных и функциональных тестов.

2 этап — контейнеризация и развертывание

Разработчики используют второй этап для сборки контейнеров своих сервисов, позволяющих использовать более глубокие

уровни тестирования, включая развертывание контейнеров для динамически созданной среды испытаний.

3 этап – атомарные функции пригодности

На третьем этапе выполняются атомарные функции пригодности для проверки масштабируемости и тестирование на проникновение. На этом этапе также запускается инструмент метрической оценки, выделяющий любой код внутри определенного пакета, который был изменен разработчиком в части возможности его тестирования. Несмотря на то что этот тест ничего не определяет, он помогает выполнению следующего этапа за счет сужения области проверки конкретного кода.

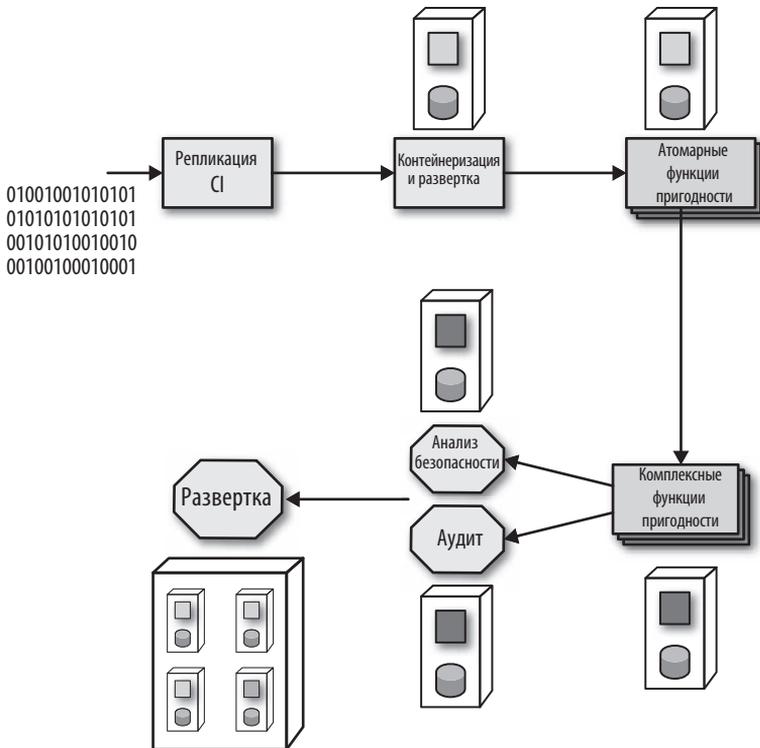


Рис. 3.8. Конвейер развертывания PenultimateWidgets

4 этап — комплексные функции пригодности

Этот этап фокусируется на применении комплексных функций пригодности, включая проверку контрактов для защиты точек интегрирования некоторых дальнейших тестов масштабируемости.

5а этап — анализ безопасности (выполняемый вручную)

Этот этап включает в себя анализ вручную, аудит и оценку любой уязвимости системы безопасности в базе кода, которые выполняются группой обеспечения безопасности предприятия. Конвейеры развертывания позволяют определить выполняемые вручную этапы, подключаемые по требованию соответствующим специалистом по безопасности.

5б этап — проведение аудита (выполняемый вручную)

Компания PenultimateWidgets зарегистрирована в городе Спрингфилде, а штат, к которому он относится, требует выполнения определенных правил проведения аудита. Группа выставления счетов встраивает этот выполняемый вручную этап в свой конвейер развертывания, который предоставляет определенные преимущества. Во-первых, аудит как функция проверки позволяет разработчикам, аудиторам и прочим специалистам единообразно рассмотреть ее поведение, что необходимо для определения надлежащего функционирования системы. Во-вторых, добавление способности к эволюционированию в конвейер развертывания позволяет разработчикам оценить влияние проектирования на поведение системы развертывания по сравнению с аналогичной автоматизированной оценкой конвейера развертывания.

Например, если анализ безопасности проводится еженедельно, а аудиторская проверка — ежемесячно, то узким местом, не позволяющим ускорить обновления, является этап аудита. Если рассматривать безопасность и аудит как этапы конвейера развертывания, решения по ним могут быть приняты более рационально: не будет ли лучше для компании повышать частоту

обновлений с помощью консультантов, чаще выполняющих необходимый аудит?

6 этап — развертывание

Последним этапом является развертка в производственную среду. Этот этап является автоматизированным для компании PenultimateWidgets, и он запускается только тогда, когда два выполняемых вручную предшествующих этапа (*анализ безопасности и аудит*) были выполнены успешно.

Заинтересованные разработчики архитектуры компании PenultimateWidgets получают автоматически создаваемые еженедельные отчеты о частоте успешных результатов/сбоев функций пригодности, которые помогают им оценивать работоспособность, порядок следования и другие факторы.

Разработка, основанная на гипотезах и на данных

Случай реструктуризации архитектуры, приведенный на с. 73, в котором используется фреймворк *Scientist*, является примером *разработки на основе данных*, которая позволяет данным управлять изменениями и концентрировать усилия на изменении технической архитектуры. Аналогичный подход, который использует предметную область, а не технические аспекты, носит название *разработки на основе гипотез* (hypothesis-driven development).

За неделю до нового 2014 года Facebook столкнулся с проблемой: за эту неделю в Facebook было выложено фотографий больше, чем фотографий в Flickr за весь год, и более миллиона фото в Facebook были помечены как оскорбительные. Facebook предоставила возможность пользователям отметить те фотографии, которые, как им кажется, могут быть оскорбительными, а затем проанализировать их, чтобы объективно установить, действительно ли это так. Такое резкое увеличение количества фотографий создало проблему: для просмотра фотографий не хватало сотрудников.

К счастью, у Facebook есть современные DevOps и возможность проводить эксперименты с пользователями. Когда спросили о вероятности того, что типичный пользователь Facebook будет вовлечен в эксперимент, один из инженеров Facebook заявил: «О, на все сто процентов — мы обычно проводим одновременно двадцать экспериментов». Инженеры использовали возможность проводить эксперименты, чтобы опросить пользователей, *почему* эти фотографии считались оскорбительными, и обнаружили много удивительных причуд в поведении людей. Например, людям не нравится признавать, что они плохо получились на фотографии, но они охотно признают, что фотограф сделал свою работу плохо. Экспериментируя с разными фразами и вопросами, инженеры могли попросить пользователей указать, почему они выделили ту или иную фотографию как оскорбительную. За относительно короткое время Facebook убрал достаточное число ложных результатов и восстановил результаты до уровня управляемой проблемы, построив платформу, позволяющую экспериментировать.

В книге *Бережливое предприятие* Барри О'Рэйлли (Lean Enterprise, Barry O'Reilly, 2014) описывает современный процесс *развития на основе гипотез*. В этом процессе вместо сбора формальных требований и траты времени и ресурсов для встраивания функций в приложения команды должны использовать научный метод. После того как команда создала версию минимально жизнеспособного приложения (нового или в результате проведения обслуживания существующего приложения), они могут в процессе обдумывания новой функции создавать гипотезы, а не устанавливать требования. Гипотезы, лежащие в основе метода разработки, формулируются для проверки того, какие эксперименты могут получить требуемые результаты и какие нарушения гипотез имеют значение для разработки будущего приложения.

Например, вместо изменения размера изображения товара на странице каталога, которое было предложено специалистом-аналитиком, считавшим это хорошей идеей, сформулируйте это предложение в качестве гипотезы: если мы сделаем изображения товаров для продажи больше, можно предположить, что это приведет к 5 %-ному увеличе-

нию продаж этих товаров. После этого проведем эксперименты с A/B-тестированием — одна группа будет с большими изображениями, а другая — с обычными. Затем сравним результаты.

Даже гибкие проекты с заинтересованными в этой предметной области пользователями постепенно могут оказаться в невыгодных условиях. Отдельное решение специалиста-аналитика может иметь смысл само по себе, но в сочетании с другими характеристиками может в конечном счете испортить весь эксперимент. В отличном практическом примере (<http://bit.ly/hypothesis-driven-ux-design>) компания mobile.de следовала логическому пути добавления новых функций в случайном порядке до того момента, когда продажи стали уменьшаться, отчасти потому, что интерфейс стал запутанным. Это часто является результатом продолжения разработки на уже сложившихся программных продуктах. Существует несколько различных философских подходов, в которых больше листингов, лучше расставлены приоритеты или лучшее разделение на группы. Чтобы помочь им сделать выбор, было построено три версии пользовательского интерфейса и принимать решение было доверено самим пользователям.

Механизмом, управляющим методами динамического программного обеспечения, является вложенный контур обратной связи: тестирование, непрерывная интеграция, итерации и т. п. Часть цепи обратной связи, которая объединяет конечных пользователей приложения, использует команды, работа которых скрыта от конечных пользователей. Используя разработку, основанную на гипотезах, мы можем беспрепятственно включать пользователей в процесс разработки и на основе их поведения строить то, что действительно ценно.

Разработка на основе гипотезы требует координации многих подвижных частей: эволюционирующей архитектуры, современной DevOps, измененного подбора требований и возможности одновременного выполнения нескольких версий приложения. Архитектура на основе сервисов (таких, как микросервисы) обычно обеспечивает параллельную работу нескольких версий с помощью продуманной маршрутизации сервисов. Например, один пользователь может выполнять приложение, используя определенные группировки

сервисов, в то время как по другому вызову приложения может использоваться совершенно другой набор экземпляров тех же сервисов. Если большинство сервисов включают выполнение многих экземпляров (например, для масштабирования), становится просто сделать некоторые из этих экземпляров несколько отличающимися повышенной функциональностью и направлять некоторых пользователей к этим экземплярам.

Эксперименты должны проводиться достаточно долго, чтобы дать значимые результаты. В общем случае, предпочтительно найти количественный способ оценки для определения лучших результатов, а не раздражать пользователей такими вещами, как всплывающие опросы. Например, может ли один гипотетический поток операций позволить пользователю завершить задание с наименьшим числом нажатий клавиш и кликов? С помощью тихого включения пользователей в процесс разработки и цепь обратной связи процесса проектирования можно построить гораздо более функциональное программное обеспечение.

Практический пример: что портировать?

Одно из приложений PenultimateWidgets, на протяжении десяти лет игравшее роль рабочей лошади, было разработано как приложение Java Swing с непрерывно увеличивающимся числом новых функций. Компания решила портировать его в веб-приложение. Но аналитики оказались перед трудным выбором: сколько существующей разрастающейся функциональности они должны перенести? И, в частности, в каком порядке они должны портировать функции нового приложения, чтобы быстрее развернуть большую часть функциональности?

Один из архитекторов из PenultimateWidgets спросил аналитиков, какие функции у приложения наиболее популярны, и аналитики этого не знали! Несмотря на то что они в течение нескольких лет определяли детали этого приложения, они не понимали, как пользователи используют его. Чтобы узнать это у пользователей, разработчики выпустили новую унаследованную версию приложения с включенным

журналированием, позволяющим отслеживать, какие пункты меню выбирали пользователи.

Через несколько недель они собрали результаты, которые дали превосходную план-схему, указывающую на то, какие именно функции следует портировать и в каком порядке. Они установили, что выставление счетов и поиск заказчика были наиболее часто используемыми функциями. Удивительно, один из подразделов этого приложения потребовал много усилий на разработку и затем редко использовался, что заставило команду отказаться от этой функции в новом веб-приложении.

4

Архитектурная связанность

Рассмотрение архитектуры часто сводится к связанности (coupling), то есть к описанию того, как соединены между собой части архитектуры и как они зависят друг от друга. Многие архитекторы считают связанность неизбежным злом, но построить сложное программное обеспечение, не опираясь на остальные компоненты, достаточно проблематично. Эволюционирующая архитектура акцентируется на надлежащей связанности — как установить, какие области архитектуры должны быть связаны между собой для обеспечения максимальных преимуществ при минимальных издержках и затратах.

Модульность

Прежде всего, давайте уточним некоторые часто используемые в архитектуре термины. Различные платформы предлагают разные механизмы повторного использования кода, но все они поддерживают некоторый способ группировки связанного кода *в модули*. *Модульность* (modularity) описывает логическую группировку связанного кода. В свою очередь, модули могут компоноваться различными физическими способами. *Компонентами* являются физические компоновки пакетов модулей. Модули подразумевают *логическую* группировку, в то время как компоненты подразумевают *физическое* разделение.

Разработчики считают полезным дальнейшее разделение *компонентов* на части на основе практики проектирования, включающей сообра-

жения по разработке и развертыванию. Одним из типов компонентов является *библиотека*, которая обычно выполняется по тому же адресу памяти, что и вызываемый код, и осуществляет связь с помощью механизмов языка вызова функций. Библиотеки обычно оказывают влияние во время компиляции. Основные проблемы, связанные с библиотеками, относятся к архитектуре приложений, поскольку самые сложные приложения состоят из различных компонентов. Другой тип компонентов, получивший название *сервисы*, обычно выполняется в собственном адресном пространстве и осуществляет связь с помощью сетевых протоколов нижнего уровня, таких как TCP/IP, или более высокого уровня, таких как простые протоколы доступа к объектам (SOAP — simple object access protocol) или протоколы передачи состояния представления (REST — representational state transfer). Проблемы, связанные с сервисами, чаще всего возникают в архитектуре интеграции приложений, делая их зависимыми от времени выполнения.

Все механизмы модульности облегчат повторное использование кода, и поэтому разумно повторно использовать код на всех уровнях, от отдельных функций и вплоть до инкапсулированных бизнес-платформ.

Квант и гранулярность архитектуры

Системы программного обеспечения связаны между собой по-разному. Как архитекторы, мы анализируем программное обеспечение, используя много разных перспектив. Но связь на уровне компонентов является не единственной связью, соединяющей программное обеспечение в единое целое. Многие концепции развития бизнеса семантически соединяли отдельные части системы вместе, создавая *функциональное сцепление* (functional cohesion). Для успешного эволюционирования программного обеспечения разработчики должны учитывать *все* точки связи, которые могут быть нарушены.

Согласно определению в физике, *квантом* является мельчайшая часть физического объекта, принимающего участие во взаимодей-

ствии. *Квант архитектуры* (architectural quantum) представляет собой независимо развертываемый компонент с высокой степенью функциональной связанности, который включает все структурные элементы, необходимые системе для надлежащего функционирования. В случае монолитной архитектуры квантом является все приложение; в этой архитектуре все имеет высокую степень связанности, и поэтому разработчики должны развертывать ее как единое целое.

**ОГРАНИЧЕННЫЙ КОНТЕКСТ, ОПРЕДЕЛЯЕМЫЙ ПРЕДМЕТНОЙ
ОБЛАСТЬЮ ПРОЕКТИРОВАНИЯ**

Книга Эрика Эванса *Предметно-ориентированное проектирование* оказала глубокое влияние на современные представления об архитектуре. Предметно-ориентированное проектирование (DDD — Domain-driven design) представляет собой метод моделирования, который позволяет осуществлять организованную декомпозицию сложных проблем предметной области. DDD определяет ограниченный контекст, в котором все, что связано с областью, можно видеть внутри этой области, но оно невидимо для других ограниченных контекстов. Перед тем как приступить к DDD, разработчики ищут всеобщее повторное использование всех объектов, принадлежащих организации. Однако создание и совместное использование артефактов приводит к массе проблем, таких как связанность, сложная координация и возрастающая сложность. Концепция *ограниченного контекста* признает, что каждый объект лучше всего работает в пределах локализованного контекста. Поэтому, вместо создания объединенной категории клиента всей организации, каждая предметная область может создать свою собственную категорию и согласовывать различия в точках интеграции. DDD оказало влияние на несколько современных стилей архитектуры, наряду со связанными факторами, такими как организация команд (описана во врезке «Знакомство с PenultimateWidgets и с их обратным законом Конвея» на с. 36 в главе 1).

В противоположность этому архитектура микросервисов определяет физически ограниченный контекст между элементами архитектуры,

формируя в пакет данных все части, которые могут измениться. Этот тип архитектуры предназначен для обеспечения возможности инкрементных изменений. В архитектуре микросервисов ограниченный контекст является квантом границы и включает в себя зависимые компоненты, такие как серверы баз данных. Он может также включать такие компоненты архитектуры, как поисковые системы и инструменты составления отчетов, все то, что способствует предусмотренной функциональности сервисов, как показано на рис. 4.1.

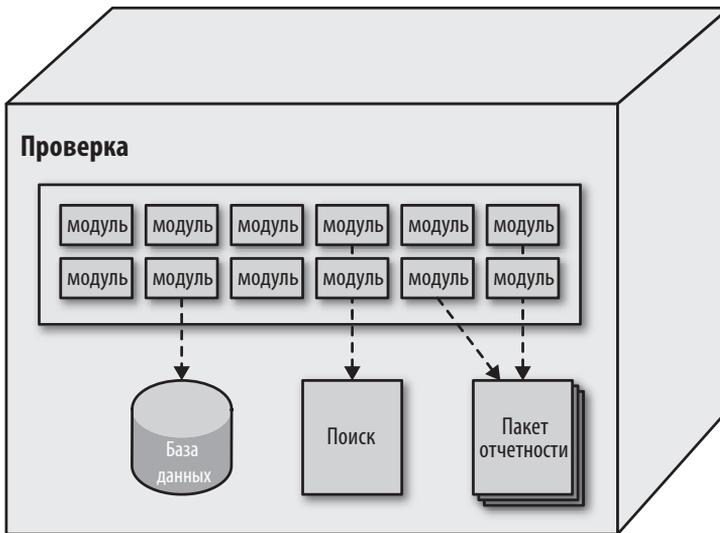


Рис. 4.1. Квант архитектуры микросервисов включает в себя эти сервисы и все зависящие от них части

На рис. 4.1 приведен сервис с компонентами кода, сервером баз данных и компонентами поисковой системы. Часть ограниченного контекста микросервисов приводит в действие все части сервиса, широко используя современные практики DevOps. В следующем разделе мы рассмотрим некоторые распространенные паттерны архитектуры и границы их типичных квантов.

Традиционно обособленные роли, такие как разработчик архитектуры и группы эксплуатации, в эволюционирующей архитектуре

должны координироваться. Архитектура является весьма абстрактной, пока не приведена в действие; разработчики должны обращать внимание на то, как их компоненты согласуются друг с другом в реальном мире. Независимо от того, какой паттерн архитектуры выбрал разработчик, он должен также явно задать размер кванта паттерна. Небольшой квант предполагает более быстрые изменения из-за небольшого масштаба. В общем случае, с небольшими частями работать легче, чем с большими. Размер кванта определяет нижнюю границу шага инкрементных изменений, допустимых в рамках той или иной архитектуры.

Как и в физике, в природе существуют четыре основных типа взаимодействия: *гравитационное, электромагнитное, сильное и слабое*. Представляющие *сильные взаимодействия* ядерные силы удерживают атомы (и поэтому обычную материю) вместе. При разрыве этих сил выделяется много энергии деления ядра. Аналогичным образом, некоторые компоненты архитектуры чрезвычайно сложно разбить на мелкие части. Образно говоря, они связаны между собой силами ядерного взаимодействия. Одним из ключей к успешной разработке эволюционирующих архитектур является определение естественной гранулярности компонентов и связей между ними для согласования возможностей, которые они должны поддерживать посредством архитектуры программного обеспечения.

В эволюционирующих архитектурах разработчики имеют дело с *архитектурными квантами*, частями системы, удерживаемыми вместе связями, которые трудно разорвать. Например, транзакции действуют как сильное взаимодействие, связывая вместе отдельные части. Несмотря на то что разработчики могут разбить на части контекст транзакции, это сложный процесс, который часто ведет к случайным усложнениям, таким как распределенные транзакции. Аналогичным образом, части предметной области могут быть сильно связаны, и разбивать приложение на более мелкие архитектурные компоненты может оказаться нежелательным.

На рис. 4.2 приводится связь между этими понятиями.

ЛИСТИНГ МОНОЛИТНОЙ АРХИТЕКТУРЫ

В течение нескольких лет мы работали над одним проектом, ориентированным на автомобильные аукционы. Неудивительно, что одним из крупных классов системы был класс `Listing`, который превратился в монстра. Разработчики предприняли несколько попыток технического рефакторинга, чтобы найти способы разбить на составляющие крупные классы, так как их использование вызывало проблемы с координацией. В конце концов была разработана схема, позволяющая разбить одну из ключевых частей, `Vendor`, на классы. Несмотря на то что технический рефакторинг был успешным, возникали проблемы между взаимодействиями разработчиков и аналитиков: разработчики продолжали сообщать об изменениях `Vendor`, который не был отдельным объектом их области. Разработчики нарушали то, что Эрик Эванс в DDD называл *единым языком* (ubiquitous language) проекта, а именно, они не обеспечили того, чтобы все термины, используемые командами, означали одно и то же. Несмотря на то что это делало некоторые вещи более удобными для разработчиков при разделении функциональности на части, семантическая связь, определяющая бизнес-процесс, была нарушена, осложнив нашу работу.

В конце концов мы отказались от рефакторинга класса `Listing` в одиночный большой объект, потому что проект разработки программного обеспечения вращался вокруг него. Мы решили проблему координации, по-другому рассмотрев `Listing`. Изменения `Listing` привели к тому, что сервер непрерывной интеграции автоматически генерировал сообщение заинтересованным командам для поощрения агрессивной интеграции. Таким образом, мы решили проблему координации с помощью инженерной практики, а не путем изменения архитектуры программного обеспечения.

Как показано на рис. 4.2, наиболее удаленным контейнером является *квант*: развертываемый модуль, который содержит все средства, включая данные, необходимые системе для надлежащего функционирования. Внутри кванта существуют несколько *компонентов*, каждый из которых состоит из кода (классов, пакетов, пространства

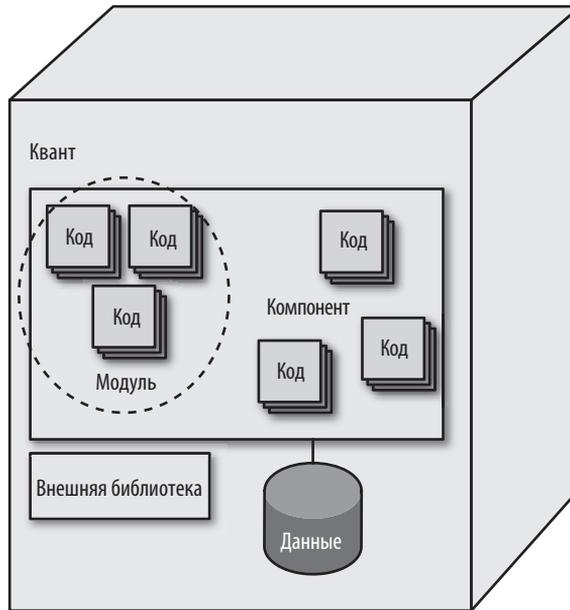


Рис. 4.2. Связь между модулями, компонентами и квантами

имен, функций и т. п.). В качестве внешнего компонента (из проекта свободного доступа) используется *библиотека*, представляющая собой пакет компонентов для повторного использования на имеющейся платформе. Конечно же, разработчики могут смешивать и подбирать все возможные комбинации этих строительных блоков.

Эволюция архитектурных стилей

Архитектура программного обеспечения существует, по крайней мере, частично, чтобы позволить эволюционировать определенным ее областям. Одной из причин использования архитектурных паттернов является возможность легкого изменения. Разные паттерны архитектуры имеют различные размеры кванта, что влияет на их способность эволюционировать. В этом разделе мы рассмотрим несколько популярных паттернов архитектуры и оценим характерный размер их кванта, а также влияние этого размера на способность архитектуры

эволюционировать, основываясь на трех критериях эволюционирования: постепенное изменение, функция пригодности и надлежащая связанность.

Обратите внимание: несмотря на важность паттерна архитектуры для успешного эволюционирования, этот фактор не является единственным определяющим фактором. Собственные характеристики паттерна должны быть объединены с дополнительными характеристиками, определенными для системы, чтобы в полной мере оценить способность области архитектуры эволюционировать.

Большой комок грязи

Прежде всего, рассмотрим случай вырождения хаотической системы без видимых признаков наличия архитектуры, известный как антипаттерн большой комок грязи¹. Несмотря на то что типичные элементы архитектуры, такие как фреймворки и библиотеки, могут существовать, разработчики не занимались разработкой целенаправленной структуры. Эти системы имеют высокую степень связанности, что ведет к появлению побочных эффектов, когда происходит изменение. Разработчики создали высокосвязанные классы с низким уровнем модульности. Схема базы данных пробралась в пользовательский интерфейс и другие части системы, эффективно изолируя их от изменений. Администраторы баз данных в течение последнего десятилетия избегали рефакторинга путем сшивания промежуточных таблиц, через которые проходит соединение. Вероятно, под влиянием драконовых ограничений бюджета группа эксплуатации собирает как можно больше систем вместе и устанавливает связь между выполняемыми операциями.

На рис. 4.3 показана схема связанности классов, которая иллюстрирует большой комок грязи: каждый узел представляет определенный класс, линии — связи (внутренние или внешние), а толщина линий указывает на число соединений.

¹ https://ru.wikipedia.org/wiki/Большой_комко_грязи

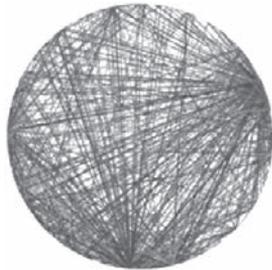


Рис. 4.3. Афферентные и эфферентные связи для дисфункциональной архитектуры

Изменения любой части приложения, указанные на рис. 4.3 (взято из реального проекта), приводят к значительным проблемам. Поскольку между классами существует множество связей, практически невозможно изменить одну часть приложения, не затронув другие. Поэтому с точки зрения способности к эволюционному развитию у такой архитектуры мало шансов. Разработчики, которым хотелось бы изменить доступ к данным во всем приложении, должны будут выявлять все места, в которых этот доступ имеется, и изменять их, рискуя при этом пропустить некоторые из этих мест.

С точки зрения эволюции эта архитектура не удовлетворяет ни одному из критериев:

Инкрементное изменение

В эту архитектуру сложно вносить какие-либо изменения. Соответствующий код разбросан по всей системе, поэтому изменение одного компонента вызовет неожиданный сбой в других. Устранение этих сбоев приведет к появлению новых сбоев, то есть вызовет лавинообразный эффект, который никогда не завершится.

Управляемое изменение с функциями пригодности

Разработка функций пригодности для этой архитектуры затруднена, поскольку в ней нет четко определенного разделения на части. Для разработки функций пригодности необходимо установить нуждающиеся в защите части архитектуры, при этом

в этой архитектуре не существует никаких функций и классов вне категории низкого уровня.

Надлежащая связанность

Этот тип архитектуры является хорошим примером *ненадлежащей связанности*. В результате разработки программного обеспечения, подобного этому, никаких архитектурных преимуществ не появляется.

В этом состоянии изменения вносить трудно и затратно. Главным образом потому, что каждая часть системы связана со всеми остальными частями, а квантом в этом случае является вся система.

Монолитная архитектура

Монолитная архитектура часто содержит значительную часть кода с высокой степенью связанности. Мы исследовали несколько вариантов архитектуры этого типа, основываясь на ее организации.

Бесструктурные монолиты

Этот паттерн архитектуры имеет несколько различных вариаций, включая системы с координацией в основном независимых классов, как это видно на рис. 4.4.

На рис. 4.4 различные модули независимо выполняют разные задания, совместно используя классы для общей функциональности. Недостаток когерентной всеохватывающей структуры затрудняет изменения в этой архитектуре.

Инкрементные изменения

Большой размер кванта затрудняет инкрементные изменения из-за высокой степени связанности, требующей развертывания больших блоков приложения. Развертывание единичного компонента затруднительно, потому что каждый компонент сильно связан с остальными, что приводит к необходимости менять также и эти компоненты.

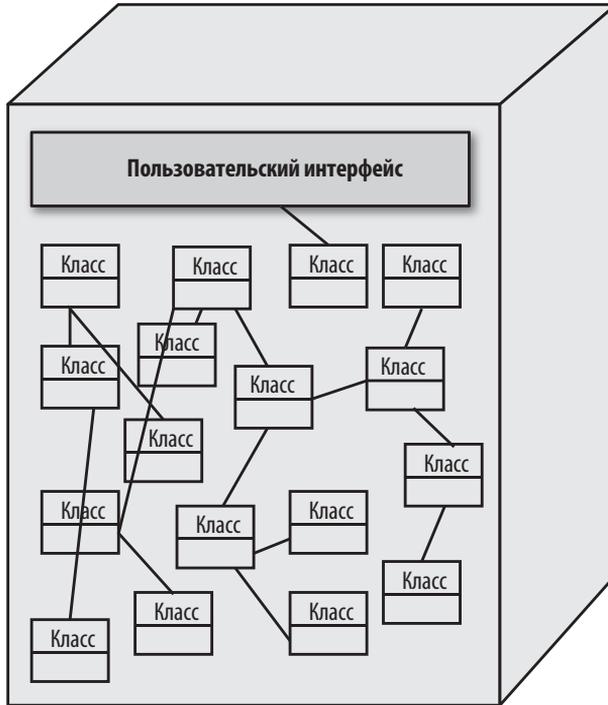


Рис. 4.4. Монолитные архитектуры иногда содержат несколько свободно связанных классов

Управляемое изменение с функциями пригодности

Разработка функций пригодности для монолитных архитектур затруднительна, но возможна. Поскольку этот тип архитектуры существует уже длительное время, появилось много инструментов и тестов, которые можно использовать для создания функций пригодности. Однако распространенные цели управляемых изменений, такие как производительность и масштабируемость, традиционно являются ахиллесовой пятой монолитных архитектур. Несмотря на то что разработчики без труда понимают монолитную архитектуру, разработка масштабируемых архитектур с высокой производительностью достаточно затруднительна, в основном из-за присущей таким архитектурам связанности.

Надлежащая связанность

Монолитная архитектура с небольшой внутренней структурой вне простых классов проявляет связанность почти так же плохо, как и архитектура типа большой комок грязи. Таким образом, изменения в части кода могут иметь непредвиденные побочные эффекты, проявляющиеся иногда в отдаленных частях базы кода.

Несмотря на то что способность эволюционировать у этой архитектуры соответствует умеренной версии архитектуры большой комок грязи, описанной на с. 93, такая архитектура может достаточно просто ухудшиться, поскольку у нее есть только несколько структурных ограничений, чтобы предотвратить это.

Слоистая архитектура

Некоторые монолитные архитектуры используют для создания слоистой архитектуры более структурированный метод, один из вариантов которого представлен на рис. 4.5.

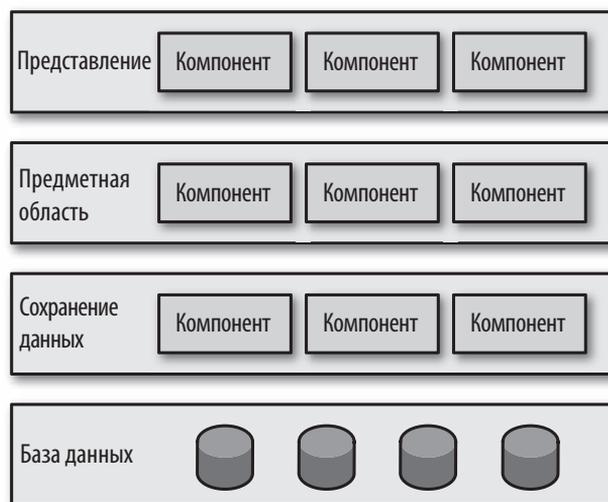


Рис. 4.5. Типичная слоистая монолитная архитектура

На рис. 4.5 каждый слой представляет собой определенные технические возможности, позволяющие разработчикам легко менять функциональность архитектуры. Основным критерием проектирования слоистой архитектуры является разделение различных технических возможностей по слоям, каждый из которых несет определенную ответственность. Основное преимущество этой архитектуры состоит в *изоляции* и *разделении функций*. Каждый слой изолирован от остальных, доступ к которым осуществляется через хорошо определенный интерфейс. Это дает возможность использовать изменения внутри слоя, не затрагивая остальных слоев и группируя сходный код вместе, оставляя пространство, необходимое для специализации и разделения слоев друг от друга. Например, слой хранения данных обычно формирует все детали того, как хранятся данные, давая возможность другим слоям игнорировать эти детали.

Во всех случаях монолитной архитектуры квантом является приложение, включающее зависимые компоненты, такие как серверы баз данных. Эволюция систем с квантами большого размера затруднительна.

Инкрементные изменения

Разработчики могут без труда вносить изменения, особенно если эти изменения ограничиваются существующими слоями. Межслойные изменения могут привести к проблемам координации, особенно если участники команды отражают слои архитектуры (см. закон Конвея на с. 33). Например, команда может передать один фреймворк хранения данных с небольшими повреждениями другим командам, потому что они могут выполнить эту работу помимо хорошо определенного интерфейса. С другой стороны, если в предметной области необходимо что-то изменить, например отгрузку заказчику, это изменение может повлиять на все слои, что требует координации.

Управляемое изменение с функциями пригодности

Оказалось, что разработчикам легче создавать функции пригодности в более структурированной версии монолитной архитектуры,

потому что такая структура архитектуры более очевидна. Разделение функций между слоями также позволяет разработчикам использовать большее число находящихся в изоляции частей, что облегчает создание функций пригодности.

Надлежащая связанность

Одно из достоинств монолитной архитектуры состоит в легкости ее понимания. Разработчики, понявшие такие концепции, как паттерны проекта, могут легко применить эти знания к слоистым архитектурам. Большая часть доступности понимания определяется удобным доступом ко всем частям кода. Слоистая архитектура позволяет легко эволюционировать техническим частям архитектуры, задаваемым слоями. Например, хорошо спроектированная (и реализованная) слоистая архитектура облегчает передачу баз данных, бизнес-правил или любой другой области с минимальными побочными эффектами.

Монолитные архитектуры обычно обладают высокой связанностью, как умышленной, так и непреднамеренной. Если разработчики используют слоистую архитектуру для разделения функций (например, используя слой хранения данных для упрощения доступа к данным), то слой обычно проявляет высокую внутреннюю и низкую внешнюю связанность. В пределах каждого слоя любой компонент взаимодействует при достижении одной цели, поэтому они стремятся к высокой связанности. В противоположность этому, разработчики обычно определяют более тщательно границу раздела между слоями, создавая низкую связанность.

Модульные монолитные архитектуры

Многие преимущества, такие как изоляция, независимость, инкрементные изменения, которые можно достичь в монолитной архитектуре, разработчики позаимствовали из микросервисов. Обратите внимание, что необходимая для этого подготовленность разработчиков должна распространяться на техническую архитектуру, включая и другие области, особенно базы данных. Современные инструменты

делают повторное использование кода настолько удобным, что разработчики ведут борьбу, чтобы достичь надлежащей связанности в тех средах, где связанность легко установить. Функции пригодности, аналогичные приведенным в примере 4.1 на с. 136, дают возможность разработчикам встроить в свои конвейеры развертывания безопасные узлы, чтобы поддерживать неизменными зависимости компонентов монолитной архитектуры.

Большинство современных языков дают возможность строить строгие правила видимости и соединений. Если разработчики архитектуры и программного обеспечения создают модульную монолитную архитектуру, применяя эти правила, у них будет в значительной степени более приспособляемая архитектура, как это показано на рис. 4.6, где представлена модульная монолитная архитектура.

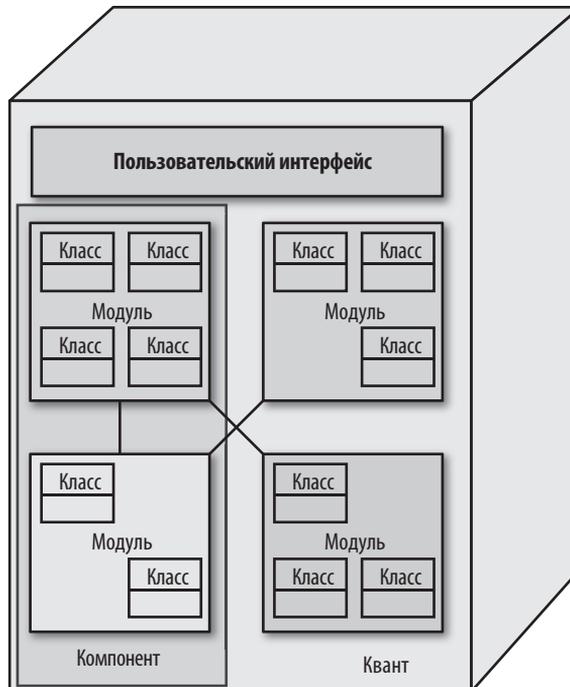


Рис. 4.6. Модульная монолитная архитектура использует логическое группирование функциональности с хорошо определенной изоляцией модулей друг от друга

Инкрементные изменения

В архитектуре этого типа можно без труда выполнять инкрементные изменения, потому что разработчики могут принудительно применять модульность. Однако, несмотря на логическое разделение функций в модулях, если компоненты содержат модули, их сложно развертывать по отдельности, так как размер кванта все еще большой. В модульной монолитной архитектуре степень развертываемости компонентов определяет скорость инкрементных изменений.

Управляемое изменение с функциями пригодности

Тесты, метрики и другие механизмы функций пригодности легче проектировать и внедрять в такую архитектуру благодаря разделению компонентов, что позволяет облегчить имитацию и другие методы тестирования, которые опираются на изоляцию слоев.

Надлежащая связанность

Хорошо спроектированная модульная архитектура является удачным примером надлежащей связанности. Каждый компонент функционально связан с четко определенными границами между ними и с низкой степенью связанности.

Монолитные архитектуры, особенно слоистые архитектуры, являются широко распространенным вариантом, используемым в начальной стадии проекта, потому что разработчики без труда понимают их структуру. Однако у многих монолитных архитектур заканчивается срок службы и они должны быть заменены из-за снижения производительности, размера базы кода и многих других факторов. Текущей целью миграции монолитной архитектуры являются архитектуры типа микросервисов, которые сложнее монолитных архитектур в отношении использования сервисов и структуры хранения данных, а также практического применения, координации, транзакций и т. п. Если команда разработки столкнулась с проблемами при разработке простейшей архитектуры, как тогда заниматься решением проблем более сложной архитектуры?

Если вы не можете построить монолитную архитектуру, то почему вы считаете, что микросервисы будут решением?

— Саймон Браун (*Simon Brown*)

Перед тем как приступить к разработке реструктуризации дорогостоящей архитектуры, разработчики могут воспользоваться преимуществами улучшения уже существующей модульности. Если нет ничего более подходящего, это превосходная стартовая точка для более серьезной последующей структуризации.

Микроядерная архитектура

Рассмотрим другой популярный тип архитектуры — микроядерной архитектуры (*microkernel architecture*), которая используется в браузерах и в интегрированных средах разработки (*IDE — integrated development environment*). Она представлена на рис. 4.7.



Рис. 4.7. Микроядерная архитектура

Микроядерная архитектура, показанная на рис. 4.7, задает ядерную систему с интерфейсом прикладного программирования, который позволяет подключать расширения. Для этой архитектуры можно использовать квант двух размеров: один для ядерной системы, а второй — для подключаемых расширений. Разработчики архитектуры обычно проектируют ядерную систему с монолитной архитектурой,

создавая привязки к хорошо известным точкам для расширений. Подключаемые модули обычно проектируются как автономные и независимо развертываемые. Поэтому этот тип архитектуры поддерживает позитивные, инкрементные изменения, а разработчики могут проектировать для тестируемости и облегчения функционирования системы функции пригодности. С точки зрения технической связанности разработчики архитектуры из практических соображений стремятся проектировать эти системы с низкой степенью связанности, что обусловлено сохранением независимости подключаемых модулей друг от друга.

Основная проблема, с которой сталкиваются разработчики в микроядерных архитектурах, связана с контрактами, представляющими определенную форму семантической связанности. Для выполнения полезной работы встраиваемые модули должны передавать информацию в систему ядра, а также из нее. Поскольку встраиваемые модули не требуют при работе координации друг с другом, разработчики могут сосредоточить свои усилия на информации и контроле версий ядерной системы. Например, большинство встраиваемых модулей браузера взаимодействуют только с браузером, а не с другими встраиваемыми модулями.

Более сложные микроядерные системы, такие как Eclipse Java IDE (<http://www.eclipse.org/>), должны обеспечивать поддержку связи внутри встраиваемых модулей. Ядро Eclipse не обеспечивает поддержку какого-либо определенного языка, кроме взаимодействия с текстовыми файлами. Сложное поведение проявляется через встраиваемые модули, которые передают информацию друг другу. Например, компилятор и отладчик должны координировать свою работу во время выполнения сеанса отладки. Поскольку встраиваемые модули не должны рассчитывать при работе на другие встраиваемые модули, то система ядра должна обрабатывать связь, выполняя координацию контрактов и общих задач, таких как управление версиями комплекса. Несмотря на то что изоляция уровней необходима, так как она делает систему менее зависимой от предыдущих состояний, ее обеспечение часто невозможно. Например, в случае Eclipse встраиваемым модулям для функционирования часто необходимы зависимые встраиваемые

модули, создающие другой уровень переходной зависимости, управляемой квантом архитектуры встраиваемого модуля.

Обычно микроядерная архитектура содержит реестр, который отслеживает установленные модули и поддерживаемые ими контракты. Создание явных связей между встраиваемыми модулями повышает семантическую связанность частей системы и тем самым увеличивает размер кванта архитектуры.

Несмотря на популярность микроядерной архитектуры из-за таких инструментов, как IDE, ее можно также использовать для широкого диапазона бизнес-приложений. Например, рассмотрим страховую компанию. Стандартные бизнес-правила для рассмотрения претензий об убытках существуют во всех подразделениях компании, хотя каждое состояние может иметь особые правила. Разработка для этой системы микроядерной архитектуры позволяет разработчикам добавлять при необходимости поддержку новых состояний и обновлять поведение отдельных состояний, не влияя при этом на остальные состояния, благодаря исходной изоляции встраиваемых модулей.

Микроядерные архитектуры достаточно хорошо обеспечивают функционирование системы, если использовать ограниченные возможности, предоставляемые подключаемыми модулями. Системы с полностью изолированными подключаемыми модулями облегчают эволюцию архитектуры благодаря отсутствию связей между подключаемыми модулями; подключаемые модули, которые должны взаимодействовать, повышают связанность и тем самым затрудняют эволюционирование архитектуры. Если проектируется система с взаимодействующими встраиваемыми модулями, дополнительно потребуются функции пригодности для защиты точек интеграции, моделируемые по контрактам, установленным для пользователей (<https://martinfowler.com/articles/consumerDrivenContracts.html>).

Система ядра с микроядерной архитектурой обычно крупнее, но устойчивее, так как большинство изменений в этой архитектуре должно возникать в подключаемых модулях (иначе разработчик не сможет разделить приложение на части). Поэтому постепенные изменения

выполняются непосредственно: конвейер развертывания запускает изменение в подключаемых модулях для проверки изменений.

Архитекторы традиционно не включают зависимости данных для микроядер внутри технической архитектуры, поэтому разработчики ПО и администраторы БД должны независимо учитывать эволюционное развитие архитектуры. Рассмотрение каждого подключаемого модуля как ограниченного контекста способствует эволюционированию архитектуры, так как снижает связанность с внешними модулями. Например, если все подключаемые модули пользуются той же базой данных, что и система ядра, разработчики должны быть обеспокоены связью между подключаемыми модулями на уровне данных. Если каждый подключаемый модуль полностью независим, эта связь по данным не может возникнуть.

С точки зрения эволюции микроядра имеют много благоприятных характеристик, включая следующие.

Инкрементные изменения

После того как система ядра укомплектована, поведение системы определяется в основном подключенными модулями, а также небольшими модулями развертывания. Если подключенные модули независимы, становится легче осуществлять инкрементное изменение.

Управляемое изменение с функциями пригодности

В этой архитектуре легче создавать функции пригодности, что связано с изоляцией системы ядра от подключенных модулей. Для такой системы разработчики используют два набора функций пригодности: *основные* (core) и *подключаемые* (plug-ins). Основные функции пригодности обеспечивают защиту от изменений системы ядра, включая проблемы, возникающие при развертывании, такие как масштабируемость. Обычно тестирование подключаемых модулей проводить легче, то есть тестирование поведения области выполняется изолированно. Разработчикам может потребоваться хорошая имитация или фиктивный модуль

из версии основной системы, чтобы облегчить проверку подключенных модулей.

Надлежащая связанность

Связанность характеристик в этой архитектуре полностью определяется паттерном микроядра. Разработка независимых подключаемых модулей облегчает изменения с точки зрения связанности. Зависимые подключаемые модули затрудняют координацию. Для обеспечения надлежащей интеграции зависимых подключаемых модулей разработчики должны использовать функции пригодности.

Для сохранения ключевых характеристик эти архитектуры должны включать комплексные функции пригодности. Например, отдельные подключаемые модули могут влиять на масштабируемость системы. Поэтому разработчики должны планировать использование комплекта проверок интеграции, который должен действовать как комплексная функция пригодности. В системе с зависимыми подключенными модулями разработчики также должны использовать комплексную функцию пригодности для обеспечения совместимости контракта и сообщений.

Событийно-ориентированная архитектура

Событийно-ориентированная архитектура (EDA — event-driven architectures) обычно объединяет несколько различных систем, используя для этого очередь сообщений. Существует два распространенных способа применения архитектур этого типа: паттерн *брокер* (broker) и паттерн *посредник* (mediator)¹. В каждом паттерне предусмотрены разные возможности системы ядра, поэтому мы рассмотрим паттерн и предпосылки эволюционирования по отдельности.

Брокеры

В случае *брокерской архитектуры* ее компоненты состоят из следующих элементов.

¹ [https://ru.wikipedia.org/wiki/Посредник_\(шаблон_проектирования\)](https://ru.wikipedia.org/wiki/Посредник_(шаблон_проектирования))

Очередь сообщений

Очередь сообщений реализуется с помощью широкого разнообразия методов, таких как JMS (Java Message Service).

Иницирующее событие

Это событие запускает бизнес-процесс.

Внутрипроцессорные события

События, проходящие между процессорами событий, для выполнения бизнес-процесса.

Процессоры событий

Активные компоненты архитектуры, которые выполняют фактическую обработку бизнес-процессов.

Если работу двух процессоров необходимо координировать, они пропускают сообщения через очередь.

Типичный рабочий поток архитектуры на основе событий приведен на рис. 4.8, на котором клиент страховой компании меняет свой адрес.

Как видно на рис. 4.8, иницирующее событие приводится в движение клиентом. Первым процессом заинтересованного события является процесс клиента, который обновляет внутреннюю запись адреса. После завершения этого процесса отправляется сообщение по адресу очереди сообщений об изменениях. Оба процесса, цена и претензии, отвечают на это событие, обновляя соответствующие характеристики. Обратите внимание, что из-за услуг, необходимых для координации, эти операции могут возникать одновременно, что является ключевым преимуществом этой архитектуры. После завершения каждого процесса отправляется сообщение `Notification` в соответствующую очередь.

Архитектура с паттерном *брокер* обладает некоторыми преимуществами при разработке надежных асинхронных систем. Например, координация и обработка ошибок являются сложными процессами

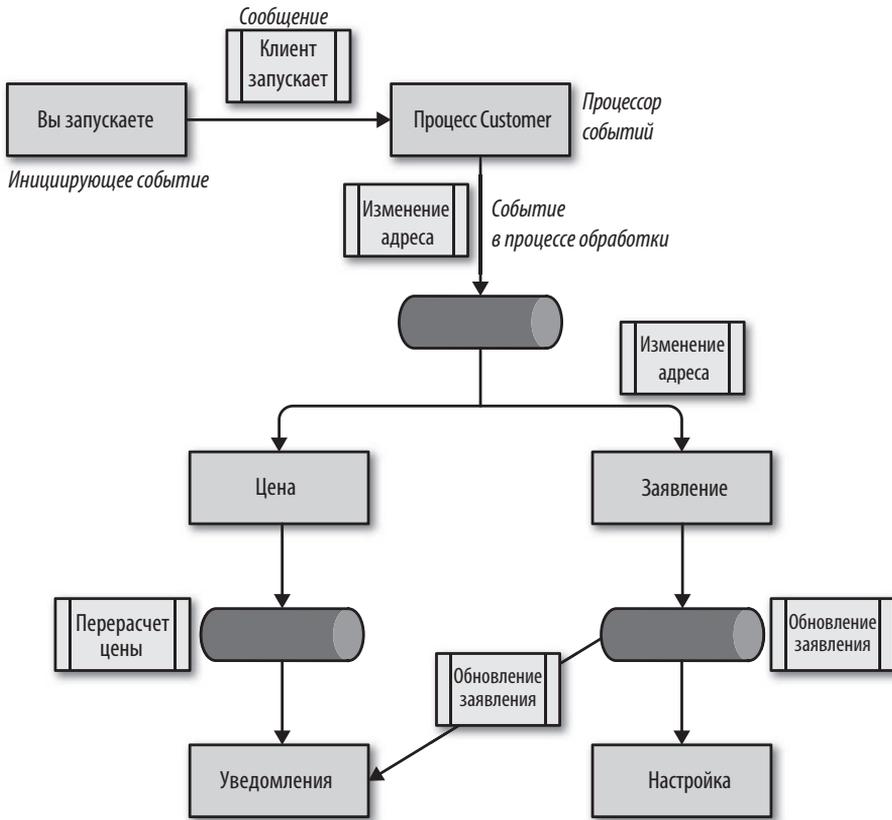


Рис. 4.8. Асинхронный рабочий поток для захвата «приводимого в движение клиентом» рабочего потока

из-за отсутствия централизованного посредника. Из-за отсутствия связей между частями архитектуры разработчики должны восстанавливать функциональную связанность обработки бизнес-процессов в этой архитектуре. Поэтому затруднены такие процессы, как транзакции.

Несмотря на проблемы с реализацией эти архитектуры чрезвычайно эволюционирующие. Разработчики могут добавлять новые процессы в систему с помощью добавлений новых обработчиков событий в существующие очереди событий, не влияя на существующие процессы. Допустим, что страховая компания хотела бы добавить систему кон-

троля ко всем обновлениям претензий. Разработчики могут добавить обработчик событий аудита в очередь претензий, не оказывая влияния на существующий рабочий поток.

Инкрементное изменение

Паттерн архитектуры *брокер* допускает инкрементные изменения в многочисленных формах. Разработчики обычно проектируют сервисы с низкой связанностью, облегчая независимое развертывание. Уменьшение связанности системы, в свою очередь, позволяет разработчикам вносить в архитектуру неразрушающие изменения. Разработка конвейеров развертывания для архитектуры с паттерном *брокер* может оказаться проблематичной из-за того, что связь в архитектуре асинхронная, и это заметно осложняет тестирование.

Управляемое изменение с функциями пригодности

Атомарные функции пригодности разработчикам легче писать в этой архитектуре из-за простоты обработки событий. Однако в этой архитектуре необходимо также использовать комплексные функции пригодности. Большая часть процессов всей системы относится к связи между отдельными сервисами, осложняя тестирование сложных рабочих потоков. Рассмотрим представленный на рис. 4.8 рабочий поток. Разработчики могут легко проверить отдельные части рабочего потока, используя блочную проверку обработки событий, но проверка всех процессов значительно более проблематичная. Существует множество способов снижения проблематичности в таких архитектурах. Например, корреляция IDS, при которой запрос помечен идентификатором, помогает отслеживать процессы, протекающие между различными сервисами. Аналогичным образом, искусственные транзакции позволяют разработчикам проверять координацию логики, фактически не вычищая код.

Надлежащая связанность

Архитектура с паттерном *брокер* обладает низкой степенью связанности, повышая способность эволюционных изменений.

Например, чтобы добавить в эту архитектуру новый процесс, в существующие конечные точки добавляются обработчики событий, не влияя при этом на существующие обработчики событий. Связанность в этой архитектуре существует между сервисами и контрактами сообщений, которые они поддерживают, представляя собой некоторую форму функционального сцепления. Функции пригодности, пользующиеся методом, аналогичным контрактам потребителя, помогают управлять точками интеграции и избегать нарушений связей.

В бизнес-процессах, которые позволяют использовать архитектуры с паттерном *брокер*, обработка событий обычно выполняется без запоминания состояний и без связи, а с помощью собственных данных, что облегчает эволюцию благодаря малочисленным проблемам с внешними связями, аналогичным тем, которые возникают с базами данных (см. главу 5).

Посредники

Другим распространенным паттерном архитектуры на основе событий является *посредник*, в котором появляется дополнительный компонент: концентратор, выступающий в роли координатора (рис. 4.9).

Посредник управляет иницилируемым клиентом событием (см. рис. 4.9) и имеет поток бизнес-процессов, который определяется: изменением адреса, пересчетом очереди, обновлением претензий, урегулированием претензий и гарантией уведомлений. Посредник направляет сообщения в очередь, которая запускает обработку соответствующих процессов. Несмотря на то что медиатор управляет координацией, эта архитектура все еще EDA (основана на событиях), что позволяет большую часть обработки выполнять одновременно. Например, процессы перерасчета очереди и обновление претензий выполняются параллельно. После завершения всех заданий посредник генерирует сообщение с целью уведомления страховой очереди для создания одиночного сообщения о состоянии. Любая обработка события, которой необходима связь с другим процессом, осуществляется через посредника. В общем случае обработки событий в этом типе

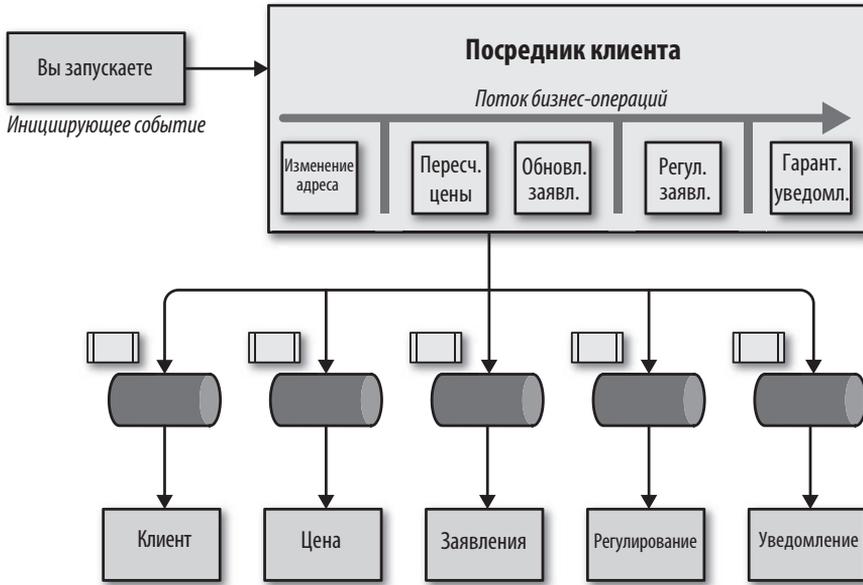


Рис. 4.9. Рабочий поток, «приводимый в движение клиентом» в архитектуре с паттерном посредника

архитектуры не приводят к вызову друг друга, потому что *посредник* определяет обход прямой связи для важной информации рабочего потока. Обратите внимание, что вертикальные черточки на рис. 4.9 указывают одновременное выполнение и координацию запросов и ответов сервиса.

Возможность координации транзакций является основным преимуществом архитектуры с паттерном *посредник*. Посредник может исключить ошибки во время обработки и создать единичное сообщение о статусе. В случае архитектуры с паттерном *брокер* этот тип координации более затруднительный. Для того чтобы создать единичное сообщение, например уведомления, координация будет использоваться при обработке сообщения-уведомления с помощью очереди сообщений, что позволит справиться с этой агрегацией. Несмотря на то что асинхронные архитектуры создают проблемы, связанные с координацией и процессами транзакций, они часто предлагают превосходную возможность выполнения одновременных операций.

Инкрементное изменение

Аналогично архитектуре с паттерном *брокер*, сервисы в паттерне *посредник* обычно небольшие и независимые. Таким образом, эта архитектура использует много преимуществ версии брокера.

Управляемое изменение с функциями пригодности

Разработчики обнаружили, что, оказывается, легче строить функции пригодности архитектуры для паттерна *посредник*, чем паттерна *брокер*. Проверки обработок отдельных событий при этом не отличаются заметным образом от паттерна *брокер*. Однако комплексные функции пригодности строить легче, потому что разработчики могут рассчитывать на посредника в управлении координацией. Например, в рабочем потоке уведомления разработчик может написать процедуру проверки и без труда сообщить, был ли весь процесс успешным, потому что посредник координирует его.

Надлежащая связанность

Несмотря на то что многие сценарии проверок легче проводить с *посредником*, при этом увеличивается связанность, затрудняя эволюционирование. *Посредник* содержит важную область логики, повышающую размер архитектурного кванта, что, в свою очередь, связывает сервисы друг с другом. Когда разработчик вносит то или иное изменение в этой архитектуре, другие разработчики должны учитывать побочные эффекты для других сервисов рабочего потока, увеличивая связанность.

С точки зрения эволюционирования архитектура паттерна *брокер* имеет очевидные преимущества, потому что в ней низкая степень связанности. В паттерне *посредник* процедура координации действует как точка связи, объединяя все задействованные сервисы вместе. В случае топологии *брокера* его характеристики могут эволюционировать в результате добавления новых процессов к существующей очереди сообщений, не влияя на остальные (за исключением случаев перегрузки очереди

трафиком, что может быть разрешено различными паттернами архитектуры и/или функциями пригодности). Поскольку топологии брокера изначально не используют связей между составляющими, обеспечить эволюционирование в этом случае легче.

Это классический пример архитектурного компромисса. Архитектура на основе событий с паттерном *брокер* предоставляет много преимуществ в отношении эволюционирования, асинхронности, масштабируемости и множества других желательных характеристик. Однако задачи общего характера, такие как координация транзакций, становятся более затруднительными.

Сервис-ориентированные архитектуры

Существуют различные архитектуры на основе сервисов (SOA — service-oriented architecture), включая множество гибридных. Ниже приводятся некоторые распространенные паттерны архитектур этого типа.

Архитектура на основе шины сервисов предприятия

Несколько лет назад стал популярен определенный способ разработки SOA, основанный на используемых сервисах и их координации с помощью шины сервисов; он получил название *шины сервисов предприятия* (ESB — Enterprise Service Bus). Шина сервисов в случае взаимодействия сложных событий действует как посредник и управляет различными рутинными операциями, такими как передача сообщений, настройка графика организации работы подразделения и т. п.

Несмотря на то что ESB-архитектуры используют такие же строительные блоки, как EDA, организация их сервисов различается и основывается на строго определенной классификации сервисов. Стили ESB в различных организациях разные, но все они основаны на разделении сервисов исходя из возможности их повторного использования, принципов совместного применения и области использования. Пример архитектуры SOA типа ESB приведен на рис. 4.10.

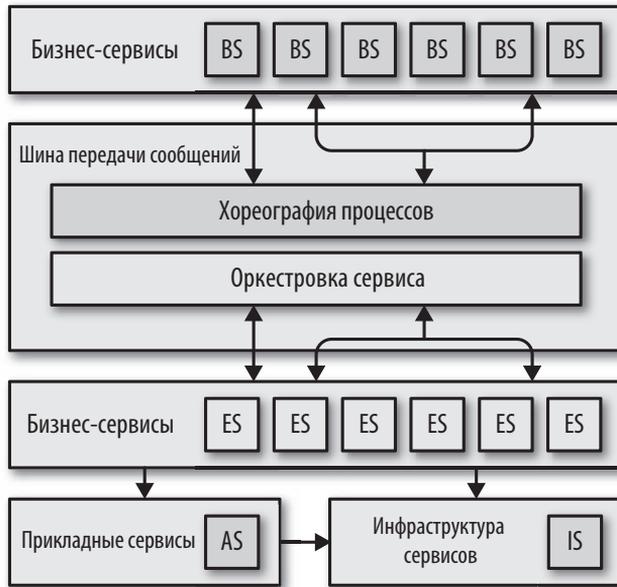


Рис. 4.10. Типичная классификация сервисов для архитектуры SOA типа ESB

На рис. 4.10 видно, что каждый слой архитектуры имеет определенные обязанности. Бизнес-сервисы определяют крупномасштабную функциональность, представляемую абстрактными понятиями, иногда определяемыми бизнес-пользователями, использующими такие стандарты, как язык описания бизнес-процессов¹ (BPEL — Business Processing Execution Language). Цель бизнес-сервисов состоит в фиксации на абстрактном уровне того, что делает компания. Чтобы знать, удалось ли определить эти сервисы на надлежащем уровне абстракции, спросите себя: «Имеем ли мы дело с...» — и ответьте утвердительно по каждому из сервисов (например, `CreateQuote` или `ExecuteTrade`). Хотя разработчик мог активировать сервис `CreateCustomer`, чтобы создать очередь, это является не основой бизнеса, а только лишь необходимым промежуточным шагом.

Абстрактные бизнес-сервисы должны вызывать код для реализации своих возможностей, которыми являются *сервисы предприятия*:

¹ <https://ru.wikipedia.org/wiki/BPEL>

определенные реализации предназначены для совместного использования командами различных сервисов. Цель таких команд — создать интеграцию повторно используемых сервисов, которые разработчик мог бы с помощью настройки «объединить» для формирования реализаций предметной области. Разработчики стремятся получить повторно используемые и спроектировать новые сервисы предприятия (чтобы понять, почему это часто не получается, см. «Антипаттерн: неправильное повторное использование кода» на с. 208).

Некоторые сервисы не нуждаются в высокой степени повторного использования. Например, одна из частей системы может нуждаться в геолокации, но это не настолько важно, чтобы выделять для этого ресурсы и делать полномасштабный сервис. *Прикладные сервисы* (application Services), приведенные в нижней левой части рис. 4.10, занимаются этими случаями. Все они ограничены определенным прикладным контекстом и не предназначены для повторного использования, относясь к определенной команде.

Инфраструктурные сервисы (infrastructure services) совместно используются сервисами, относящимися к группам инфраструктуры для управления нефункциональными требованиями, такими как мониторинг, регистрация, аутентификация/авторизация и т. п.

Характеристики архитектуры SOA версии ESB определяются архитектурным компонентом шины сообщений, который отвечает за различные задачи:

Посредничество и маршрутизация

Шина сообщений знает, как разместить и организовать связь с сервисами. Обычно она поддерживает реестр физических положений, протоколы и другую информацию, необходимую для активации сервисов.

Процесс оркестровки (orchestration) и хореографии (choreography)

Шина сообщений составляет все сервисы предприятия вместе и управляет задачами, такими как порядок выполнения вызовов.

Доработка и преобразование вызовов

Одно из преимуществ блока интеграции состоит в его способности обслуживать протокол и выполнять определенные преобразования для приложений. Например, `ServiceA` может вести связь по протоколу HTTP и вызывать `ServiceB`, который использует протокол RMI/IIOP. Разработчики могут настроить конфигурацию шины сообщений на невидимое выполнение необходимого преобразования всякий раз, когда это преобразование необходимо.

Архитектурный квант SOA типа ESB огромен! Он состоит из всей системы, в большей степени напоминая монолитную архитектуру. Но он намного сложнее, потому что является распределенной архитектурой. Выполнять единичные эволюционные изменения в архитектуре SOA типа ESB из-за систематики чрезвычайно трудно, тогда как помощь повторному использованию вредит изменениям общего характера. Рассмотрим, например, концепцию предмета `CatalogCheckout` архитектуры SOA, которая распределена по всей технической архитектуре. Для изменения только `CatalogCheckout` требуется координация тех частей архитектуры, которые находятся в разных командах, что приводит к значительным затруднениям выполнения координации.

Сравним `CatalogCheckout` с ограниченным контекстом при разделении микросервисов на части. В архитектуре микросервисов каждый ограниченный контекст представляет бизнес-процесс или рабочий поток. Поэтому разработчикам следует создавать ограниченный контекст вокруг чего-то похожего на `CatalogCheckout`. Вероятно, сервису `CatalogCheckout` потребуются дополнительные сведения о пользователе, но каждый ограниченный контекст имеет собственные объекты. Если другие ограниченные контексты также имеют понятие `Customer`, разработчики не пытаются ввести один, совместно используемый класс `Customer`, что могло бы стать предпочтительным подходом в архитектуре SOA на основе событий. Если ограниченными контекстам `CatalogCheckout` и `ShipOrder` необходимо совместное использование информации о своих пользователях, они могут выполнить это с помощью обмена сообщениями, а не пытаться установить единое представление.

Архитектура SOA на основе событий не проектировалась для эволюционирования, поэтому неудивительно, что ни одна из эволюционных областей не проявила себя успешно.

Инкрементное изменение

Несмотря на хорошо обоснованную систему технических сервисов, позволяющую повторное использование и разделение ресурсов, эта система испытывает большие затруднения, выполняя наиболее распространенные типы изменений в предметной области. Большинство команд архитектуры SOA разделены так же, как архитектура, и требуются колоссальные усилия координации для простых изменений. Архитектуру SOA на основе событий также сложно ввести в эксплуатацию. Эта архитектура состоит из множества физических развертываемых модулей, что затрудняет процесс автоматизации и координации. Никто не рискнет выбрать ESB для динамичности и простоты использования.

Управляемое изменение с функциями пригодности

В общем случае тестирование архитектуры на основе событий затруднено. Ни одна из частей архитектуры не является завершенной — все они являются частью более крупного рабочего потока и обычно не предназначены для изолированного тестирования. Например, сервис предприятия может быть предназначен для повторного использования, но тестирование его основных характеристик проблематично, потому что он является только частью разнообразных рабочих потоков. Разработка атомарных функций пригодности практически невозможна, оставляя большую часть проверок для крупномасштабных комплексных функций пригодности, которые выполняют сквозное тестирование.

Надлежащая связанность

С точки зрения возможности для предприятия повторного использования имеет смысл необычная система организации архитектуры. Если разработчики смогут ухватить сущность повтор-

ного использования для каждого рабочего потока, они, в конце концов, опишут все особенности поведения компании раз и навсегда, а разработка будущего применения будет сводиться к подключению существующих сервисов. Однако в реальном мире это возможно не всегда. Архитектура SOA на основе событий построена так, что не предусматривает независимое эволюционирование своих частей, поэтому оказывает этому процессу чрезвычайно низкую поддержку. Проектирование для повторного использования категорий нарушает возможность выполнять эволюционные изменения на уровне архитектуры.

Архитектуры программного обеспечения были созданы не в вакууме; они всегда отражают экосистему, в которой были определены. Например, когда была популярна архитектура SOA, компании не пользовались такими инструментами, как операционная система с открытым исходным кодом, — вся инфраструктура была коммерческой, лицензионной и дорогостоящей. Десять лет назад разработчик, предлагающий архитектуру микросервисов, в которой каждый сервис выполнялся на своем собственном экземпляре операционной системы и машины, посмеялся бы над центром управления, потому что соответствующая архитектура была бы чрезвычайно дорогой. Из-за динамического равновесия экосистемы разработки программного обеспечения новые архитектуры появляются из-за абсолютно новой среды.

Несмотря на то что разработчики могут все еще выбирать архитектуру SOA на основе событий для интеграции тяжелых сред, получения определенного масштаба, системы или по другим разумным причинам, они делают этот выбор не для эволюционирования архитектуры, для чего она абсолютно не подходит.

Микросервисы

Сочетание опыта проектирования непрерывного развертывания с разбиением на логические блоки ограниченного контекста создает идейную основу для микросервисной архитектуры, наряду с концепцией архитектурного кванта.

В слоистой архитектуре акцент делается на *техническую* область или на то, как работает механика приложений: хранение данных, пользовательский интерфейс, бизнес-правила и т. п. Большинство архитектур программного обеспечения акцентирует внимание именно на этих технических областях. Однако существует еще одна точка зрения. Представьте, что одним из ключевых ограниченных контекстов является приложение *проверка* (checkout). Где именно это приложение находится в слоистой архитектуре? Область понятий, таких как *проверка*, разбросана по всем слоям этой архитектуры. Поскольку эта архитектура разделена на технические слои, то в этой архитектуре нет четкого определения *области* части архитектуры, как это видно на рис. 4.11.

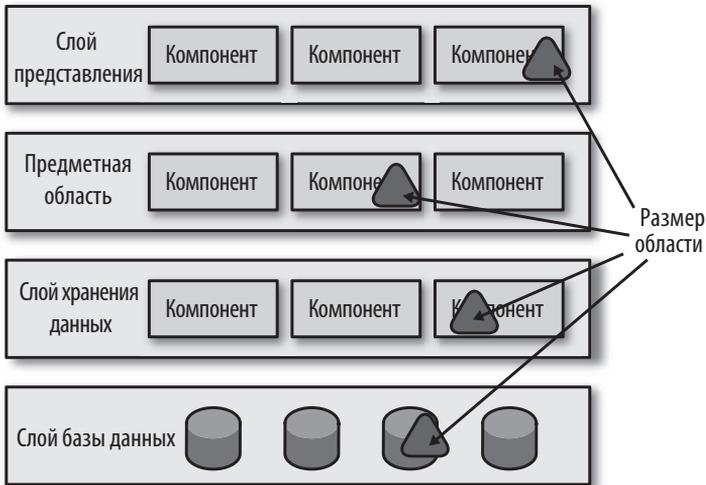


Рис. 4.11. Части области, встроенной внутри технической архитектуры

На рис. 4.11 показано, что определенная часть *проверки* существует в пользовательском интерфейсе, другая — в предметном слое, а для хранения данных задействованы нижние слои. Поскольку слоистая архитектура не предназначена для использования концепций области, разработчики вынуждены менять все слои, чтобы что-либо изменить в областях. Что касается предметной области, слоистая архитектура обладает нулевой степенью эволюционирования. В архитектуре с высокой степенью связанности изменения затруднены из-за связи

между ее частями, которые разработчик хотел бы изменить. До сих пор в большинстве проектов общий модуль изменения вращается вокруг концепций области. Если команда разработки программного обеспечения организована «анклавами», напоминающими элементы слоистой архитектуры, тогда для изменений в приложении *проверка* требуется координация многих команд.

В противоположность этому, рассмотрим архитектуру, в которой размерность *области* представляет основные подразделы архитектуры, как показано на рис. 4.12.

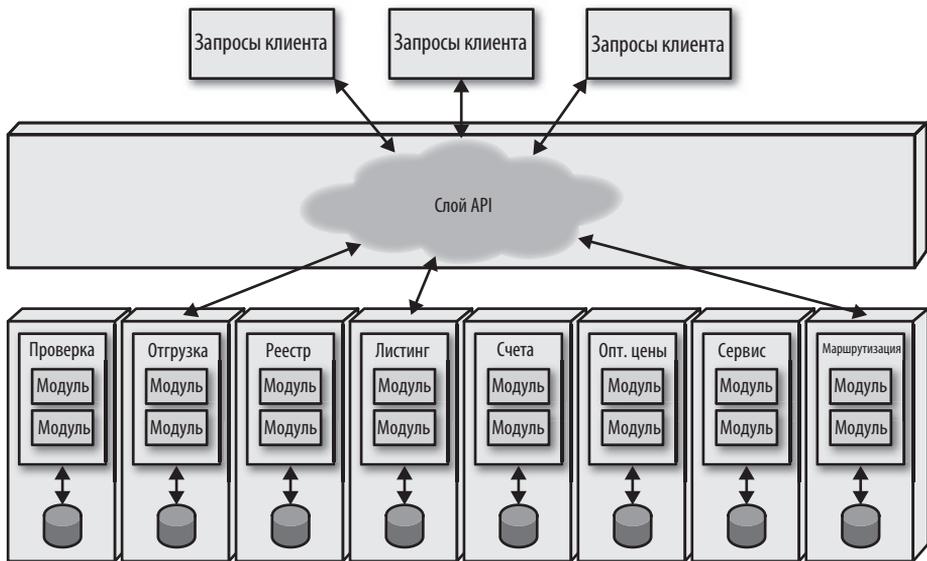


Рис. 4.12. Разделение архитектуры на микросервисы по всем областям, включающим техническую архитектуру

Как показано на рис. 4.12, каждый сервис в концепции проектирования на основе предметной области при создании архитектуры с высоким уровнем разделения формирует техническую архитектуру и все остальные зависимые компоненты (такие, как базы данных) в ограниченный контекст. Каждый сервис обладает всеми частями своего ограниченного контекста и осуществляет связь с другими ограниченными контекстами с помощью обмена сообщениями (такими,

как REST или очередь сообщений). Поэтому ни один сервис не может знать деталей реализации другого сервиса (такого, как схемы базы данных), предотвращая ненадлежащую связанность. Целью работы этой архитектуры является замена одного сервиса на другой, что не нарушает работу остальных сервисов.

Архитектуры микросервисов обычно следуют нескольким принципам, как сообщается в описании разработки архитектур микросервисов:

Моделирование предметной области

Акцент при проектировании микросервисов делается на предметную область, а не на техническую архитектуру. Поэтому квант архитектуры отражает ограниченный контекст. Некоторые разработчики делают ошибки, связанные с тем, что ограниченный контекст представляет единичный объект, такой как `Customer`; вместо этого он представляет бизнес-контекст и/или рабочий поток, такой как `CatalogCheckout`. Цель архитектуры микросервисов — посмотреть, как небольшие группы разработчиков могут сделать каждый сервис, а не создать полезный ограниченный контекст.

Скрытые детали реализации

Техническая архитектура в микросервисах сформирована внутри границ сервиса, который основывается на предметной области. Каждая область образует физически ограниченный контекст. Сервисы интегрируются друг с другом обменом сообщениями или совместными ресурсами, но не раскрытием таких деталей, как схемы баз данных.

Культура автоматизации

Архитектура микросервисов охватывает непрерывную поставку, используя для этого конвейеры развертывания с целью тщательной проверки кода и автоматизации выполнения задач, таких как инициализация и развертывание машины. Автоматическое тестирование особенно полезно в быстро меняющейся среде.

Высокая степень децентрализации

Микросервисы формируют архитектуру *без распределения ресурсов*, цель которой снизить, насколько это возможно, связанность. В общем случае для связи предпочтительно использовать дублирование. Например, оба сервиса, `CatalogCheckout` и `ShipToCustomer`, используют такое понятие, как `Item`. Поскольку обе команды имеют одно и то же имя и сходные свойства, разработчики пытаются повторно использовать их в обоих сервисах, считая, что это может сэкономить время и усилия. Вместо этого происходит увеличение усилий, потому что изменения теперь должны распространяться между двумя командами, которые совместно используют этот компонент. И при изменении сервиса разработчики должны будут учитывать изменение совместно используемого компонента. С другой стороны, если каждый сервис имеет свой собственный `Item` и передает необходимую информацию из сервиса `CatalogCheckout` в сервис `ShipToCustomer`, не используя связь с компонентом, тогда они могут изменяться независимо.

Независимое развертывание

Разработчики и группы эксплуатации ожидают, что каждый компонент сервиса будет развернут независимо от других сервисов (и другой инфраструктуры), что отражает ограниченный контекст. Возможность для разработчика развертывать один сервис, не оказывая влияния на другой, является одним из преимуществ архитектуры этого типа. Кроме того, разработчики обычно автоматизируют операции развертывания и задачи на выполнение операций, включая одновременные тестирования и непрерывную поставку.

Изолированный сбой

Разработчики изолируют сбой внутри контекста микросервисов и при выполнении координации сервисов. Каждый сервис должен обрабатывать разумные сценарии ошибок и, при возможности, восстанавливаться после их устранения. Различные

практики использования DevOps (такие, как паттерн обрыва цепи (circuit breaker pattern), перемычки (bulkheads) и т. п.) часто появляются в этих архитектурах. Многие архитектуры микросервисов соблюдают Реактивный манифест (<https://www.reactivemanifesto.org/>), перечень принципов работы и координации, которые обеспечивают более надежные системы.

Высокая степень визуализации

Разработчики не могут рассчитывать на контроль в ручном режиме сотен или тысяч сервисов (сколько сеансов многоадресной SSH-сессии может наблюдать один разработчик?). Поэтому оперативный контроль и регистрация становятся первостепенными проблемами этой архитектуры. Если группы эксплуатации не могут контролировать один из этих сервисов, они также могут и не существовать.

Основными целями микросервисов является изоляция областей с помощью физически ограниченного контекста и концентрация усилий на выявление проблемной области. Поэтому архитектурным квантом является сервис, что представляет собой превосходный пример эволюционирующей архитектуры. Если одним сервисам необходимо эволюционировать, чтобы изменить свою базу данных, ни один другой сервис при этом не будет затронут, потому что ни один сервис не может знать таких деталей реализации, как схемы. Конечно, разработчики изменений сервиса распространят эту информацию между сервисами через точку интеграции (желательно обеспеченной защитой функции пригодности, такой как *контракты пользователя*) и дадут возможность разработчикам вызывающих сервисов оставаться в блаженном неведении относительно изменений.

С учетом того, что микросервисы являются нашим примером эволюционирующей архитектуры, неудивительно, что они имеют преимущества с точки зрения эволюционирования.

Инкрементное изменение

Оба аспекта постепенных изменений легко реализуются в архитектуре микросервиса. Каждый сервис образует ограниченный

контекст вокруг понятия области, позволяя легче производить изменения, затрагивающие только этот контекст. Архитектура микросервисов сильно зависит от практики автоматизации *непрерывной поставки*, использующего конвейер развертывания и современную практику DevOps.

Управляемое изменение с функциями пригодности

Разработчики могут без труда создать атомарную и комплексную функции пригодности для архитектуры микросервисов. Каждый сервис имеет хорошо определенную границу, обеспечивая тестирование разного уровня всех компонентов сервиса. Сервис должен с помощью интеграции выполнять координацию, что также приводит к необходимости тестирования. К счастью, сложность методов тестирования растет параллельно с разработкой микросервисов.

Надлежащая связанность

Архитектура микросервисов обычно использует два типа связи: *интеграция* (integration) и *шаблон сервиса* (service template). Связь интеграции очевидна — для того, чтобы передать информацию, сервисы должны вызывать друг друга. Другой тип связи, *шаблоны сервиса*, предотвращает вредное дублирование. Разработчики и группа эксплуатации имеют преимущества, если различные функции согласованы и управляются микросервисами. Например, каждый сервис должен включать мониторинг, регистрацию, аутентификацию/авторизацию и прочие возможности связующего программного обеспечения. Если поручить их каждой команде сервиса, то обеспечение согласования и управления жизненным циклом, вероятнее всего, пострадают. Выбрав соответствующие точки соединения технической архитектуры в паттернах сервиса, команда инфраструктуры может управлять этой связью, одновременно освобождая команды отдельных сервисов от соответствующих работ. Команды, занимающиеся областями, просто расширят шаблон и запишут их поведение. При обновлении до изменений инфраструктуры шаблон автоматически подключает

эти изменения во время следующего выполнения конвейера развертывания.

Физически ограниченный контекст в микросервисх в точности коррелирует с нашей концепцией кванта архитектуры, представляя собой физически несвязный развертываемый компонент с высокой функциональной связанностью.

Одним из ключевых принципов архитектуры микросервисов является жесткое разбиение по ограниченному контексту областей. Техническая архитектура встроена внутри частей областей с соблюдением проектирования на основе ограниченных контекстов предметных областей, делая каждый сервис физически отдельным, что ведет с технической точки зрения к архитектуре *без разделения ресурсов*. Физическое разделение предполагается для каждого сервиса, что обеспечивает их легкую замену и эволюционирование. Поскольку в каждый микросервис в рамках ограниченного контекста встроена техническая архитектура, то любой сервис может эволюционировать любым образом. Поэтому области микросервисов, способные к эволюционированию, соответствуют числу сервисов, к каждому из которых разработчик может обращаться независимо, поскольку каждый сервис имеет низкий уровень связанности.

«БЕЗ РАЗДЕЛЕНИЯ РЕСУРСОВ» И НАДЛЕЖАЩАЯ СВЯЗАННОСТЬ

Архитекторы часто называют микросервисы архитектурой «без разделения ресурсов» (share nothing). Основным преимуществом этого типа архитектуры является отсутствие связанности в слое технической архитектуры. Но те специалисты, кто осуждает связанность, часто рассуждают о ненадлежащей связанности. В конце концов, система программного обеспечения без всякой связанности не способна вообще ни на что-либо. Термин «без разделения ресурсов» в действительности означает «без запутанных точек соединений». Даже в микросервисах (таких, как инструменты, фреймворки, библиотеки и т. п.) некоторые ресурсы необходимо использовать совместно и координировать. Например, регистрация, мониторинг, обнаружение сервисов и т. п. Если команда сервисов забывает до-

бавить возможность контроля, это приводит к появлению проблем во время развертывания. Если в архитектуре микросервисов невозможно контролировать сервис, она превращается в черную дыру.

Шаблоны сервисов (такие, как DropWizard, <https://www.dropwizard.io/> и Spring Boot, <http://spring.io/projects/spring-boot>) являются известными решениями этой проблемы в архитектуре микросервисов. Использование этих фреймворков дает возможность команде DevOps встраивать согласованный инструмент, фреймворки, версии и т. п. в шаблон сервиса. Команды сервисов используют шаблон, чтобы «встроить» процессы предметной области. При обновлении инструмента контроля команда разработки сервиса может координировать обновление шаблона сервиса, не беспокоя другие команды.

При наличии очевидных преимуществ почему бы разработчикам заранее не выбрать этот тип? Десять лет назад автоматическая инициализация машин была невозможна. Операционные системы были коммерческими и лицензионными с незначительным уровнем поддержки автоматизации. Ограничения со стороны реального мира, такие как бюджет, влияют на архитектуру, что является одной из причин того, что разработчики строят все больше и больше архитектур с распределяемыми ресурсами, разделенными на технические слои. Если выполняемые операции дорогостоящие и громоздкие, то вся архитектура создается вокруг них, как в случае ESB-SOA.

Система непрерывного развертывания и движение DevOps добавляет новый фактор в динамическое равновесие. Теперь осуществляемый машинами контроль и поддержка жизни версий имеют высокую степень автоматизации. Конвейеры развертывания реализуют многочисленные среды тестирования одновременно с поддержкой безопасного непрерывного развертывания. Поскольку большая часть программного стека имеет открытый исходный код, то лицензирование и другие проблемы больше не влияют на архитектуру. Сообщество среагировало на новые возможности, появившиеся в экосистеме развертывания программного обеспечения, позволяющие построить больше разных типов предметно-центричных архитектур.

В архитектуре микросервисов предметная область формирует технические и другие архитектуры, что облегчает эволюционирование. Ни одну из точек зрения на архитектуру нельзя считать «правильной»; они, скорее всего, отражают цели разработчика при создании своих проектов. Если акцент делается целиком на техническую архитектуру, тогда становится легче вносить изменения в разных областях. Однако если игнорировать перспективу предметной области, тогда эволюционирование различных областей будет не лучше, чем в случае архитектуры большого комка грязи.

Одним из основных факторов, влияющих на эволюционирование приложения на архитектурном уровне, является то, насколько *неумышленно связной* (unintentionally coupled) является каждая часть системы. Например, в слоистой архитектуре ее разработчик связывает определенным образом между собой все слои. Однако разделы предметной области при этом оказываются связными неумышленно, что затрудняет эволюционирование этих разделов, потому что архитектура спроектирована вокруг слоев технической архитектуры, а не предметной области. Поэтому одним из важных аспектов способной эволюционировать архитектуры является *надлежащее связывание между собой разделов предметной области*. В главе 8 мы обсуждаем, как определять и использовать границы кванта в практических целях.

Архитектуры на основе сервисов

Более распространенным типом архитектуры, используемым для переходов, является *архитектура на основе сервисов* (service-based architecture), которая аналогична архитектуре микросервисов, хотя и отличается от нее тремя важными характеристиками: гранулярностью сервиса, границами баз данных и промежуточным программным обеспечением, используемым для интеграции. Архитектура на основе сервисов все еще относится к предметно-центричным архитектурам, но преодолевает некоторые проблемы, с которыми сталкиваются разработчики при реструктуризации существующих приложений с целью обеспечения большей степени эволюционирования.

Более крупная гранулярность сервисов

Сервисы в этой архитектуре имеют более крупную гранулярность, большую степень монолитности архитектуры, основанные только на понятии предметной области. Несмотря на то что они все еще основаны на предметной области, более крупный размер единицы изменения (разработка, развертывание, связанность и ряд других факторов) снижает возможность изменений. Когда разработчики оценивают монолитное приложение, они часто видят более крупные элементы деления вокруг общей области, как в случае приложения `CatalogCheckout` или `Shipping`, что формирует неплохой первый проход при разделении архитектуры на части. Цели функциональной изоляции в архитектуре на основе сервисов те же, что в случае архитектуры микросервисов, но в первом случае их сложнее достичь. Поскольку размер сервиса больше, разработчики должны принимать во внимание больше точек связанности, а также сложности, присущие более крупным кускам кода. В идеале, архитектура должна поддерживать такой же тип конвейера развертывания и небольшую единицу изменения, как в случае архитектуры микросервисов: если разработчик меняет сервис, он должен подключить конвейер развертывания, чтобы перестроить зависимые сервисы, включая приложения.

Область базы данных

Архитектура на основе сервисов имеет тенденцию к созданию монолитной базы данных, независимо от того, насколько хорошо организованы сервисы. В любых приложениях недопустимо или невозможно реструктурировать схемы взаимодействующих в течение многих лет или десятилетий баз данных в мелкоштатные фрагменты, пригодные для микросервисов. Несмотря на невозможность дробить, данные в некоторых ситуациях могут оказаться непригодными, и их невозможно использовать в некоторых проблемных областях. Системы с высоким уровнем транзакций плохо подходят для микросервисов, так как из-за координации между сервисами использование архитектуры с транзакциями слишком дорого обходится. Системы со слож-

ными требованиями к транзакциям более четко отображаются в архитектурах на основе сервисов, что связано с менее строгими требованиями.

Несмотря на то что база данных остается нераспределенной, компоненты, которые используют базу данных, скорее всего, изменятся, становясь более детализированными. Поэтому, несмотря на то что соответствие между сервисами и лежащими в их основе данными может измениться, это не требует значительной реструктуризации. Мы включили проектирование эволюционирующей базы данных в главу 5.

Связующее ПО для интеграции

Третье различие между архитектурой микросервисов и архитектурой на основе сервисов относится к координации с помощью посредника, такого как сервисная шина. Разработка новых приложений микросервисов позволяет разработчикам не беспокоиться о старых точках интеграции, но все эти опасения относятся ко многим средам с прежними системами, которые все еще выполняют полезную работу. Узлы интеграции, такие как шины сервисов предприятия, отличаются формированием сцепления между различными сервисами с разными протоколами и форматами сообщений. Если разработчики оказались в среде, в которой архитектура интеграции приложений имеет высший приоритет, то использование узла интеграции облегчает добавление и изменение зависимых сервисов.

Использование узла интеграции является классическим архитектурным компромиссом: с его помощью разработчикам потребуется писать меньший объем кода для сцепления приложений, и они могут использовать его для имитации координации транзакций между сервисами. Однако применение узла повышает архитектурную связанность между компонентами, и разработчики могут больше не делать изменений независимо и без координации с другими командами. Функции пригодности могут снизить некоторые из этих затрат на координацию, но чем больше разработчики увеличивают связанность, тем труднее системе эволюционировать.

Далее описывается то, как архитектура на основе сервисов противодействует эволюционированию нашей архитектуры:

Инкрементное изменение

Инкрементное изменение в этой архитектуре считается практически функциональным, потому что каждый сервис является предметно-центричным. Большинство изменений в проектах ПО возникает вокруг областей, которые обеспечивают согласование между единицей изменения и квантом развертывания. Несмотря на то что эта архитектура не настолько динамична, как микросервисы (что связано с размером сервисов), многие преимущества микросервисов сохраняются.

Управляемое изменение с функциями пригодности

Обычно разработчикам сложнее написать функции пригодности в архитектурах на основе сервисов, чем в микросервисах, из-за повышенной связанности (чаще всего в базе данных) и более крупного ограниченного контекста. Повышенное программное сцепление часто затрудняет написание тестов, а повышенное сцепление по данным создает массу собственных проблем. Более крупный ограниченный контекст в архитектуре на основе сервисов предоставляет разработчикам больше возможностей для создания внутренних точек сцепления, одновременно усложняя тестирование и прочую диагностику.

Надлежащая связанность

Связанность часто является причиной, по которой разработчики выбирают архитектуру на основе сервисов, а не микросервисы: сложности реконструкции схем баз данных, высокая степень связанности внутри монолитной архитектуры, предназначенной для реструктуризации, и т. п. Разработка предметно-центричных сервисов помогает обеспечить надлежащую связанность, а шаблоны сервисов помогают создать надлежащий уровень связанности технической архитектуры.

Не использующая сервисов архитектура *VaaS* допускает ограниченное эволюционирование, но дает привлекательные рабочие характери-

стики. Архитектуры на основе сервисов определенно более эволюционирующие, чем архитектуры SOA на основе событий. Степень отклонения, допускаемая разработчиками от ограниченного контекста, в большой степени определяется размером кванта и тем, насколько вредоносной может оказываться связанность.

Архитектуры на основе сервисов являются удачным компромиссом между философской чистотой микросервисов и прагматичной реальностью многих проектов. За счет ослабления ограничений на размер сервиса, независимости базы данных и побочной, но полезной связанности, эта архитектура устраняет большинство болезненных аспектов микросервисов, одновременно сохраняя большую часть преимуществ.

Бессерверная архитектура

Появление бессерверной архитектуры сместило равновесие в разработке программного обеспечения в отношении двух широких понятий, оба из которых применимы к эволюционирующим архитектурам.

Те приложения, которые значительно или в первую очередь зависят от сторонних приложений и/или «облачных» сервисов, получили название BaaS (Backend as a Service — бэкенд как сервис). Рассмотрим простой пример, представленный на рис. 4.13.

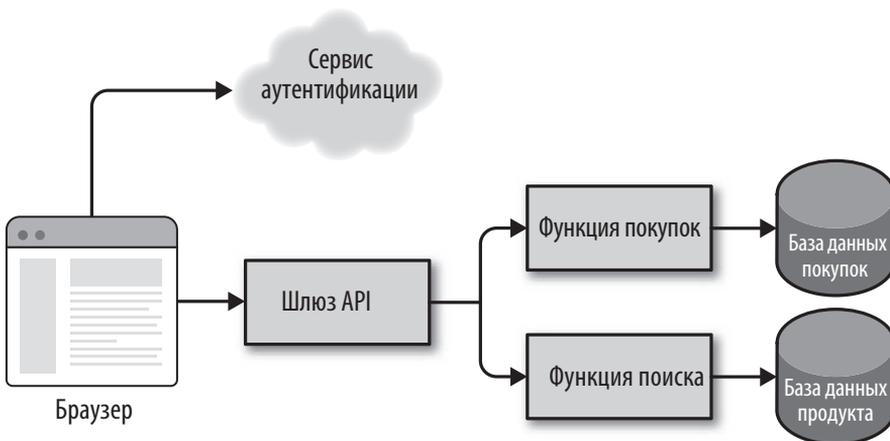


Рис. 4.13. Бессерверная архитектура BaaS

На рис. 4.13 приведен пример, в котором разработчику практически не пришлось писать код. Составление такой архитектуры сводится к совместному написанию всех сервисов, включая сервисы идентификации, передачи данных и т. п., необходимые для интеграции элементов архитектуры. Этот тип архитектуры привлекателен тем, что приходится писать всего несколько строк кода, а сохранять — еще меньше. Однако сложные для интеграции архитектуры имеют свои собственные проблемы.

Другим типом бессерверной архитектуры является FaaS (Function as a Service — функция как сервис), которая полностью отказывается от инфраструктуры (по крайней мере, с точки зрения разработчика), инициализируя инфраструктуру по запросу, автоматически управляя масштабированием, инициализацией и массой других функций управления. Функции в FaaS подключаются типами событий, которые определяет провайдер услуг. Например, веб-сервисы Amazon (Amazon Web Services, AWS) являются распространенным провайдером FaaS, обеспечивая обновление файла (в S3) подключением событий, временем (спланированными заданиями) и сообщениями, добавляемыми в шину сообщений (например, Kinesis, <https://aws.amazon.com/ru/kinesis/>). Провайдеры часто ограничивают время, которое может занимать запрос, а также вводят другие ограничения, в основном связанные с состоянием. Можно сказать, что функции FaaS не имеют состояния и перекладывают управление состоянием на вызывающее устройство.

Инкрементное изменение

Инкрементное изменение в архитектуре без сервисов должно состоять из повторного развертывания кода, и все проблемы инфраструктуры существуют за абстракцией «несерверной обработки». Архитектуры этого типа естественным образом подходят для конвейеров развертывания, чтобы управлять тестированием и постепенным развертыванием по мере того, как разработчики будут вносить изменения.

Управляемое изменение с функциями пригодности

Функции пригодности играют важную роль в архитектуре этого типа для обеспечения сохранения согласованности точек интеграции. Поскольку координация между сервисами играет ключевую роль, то ожидается, что разработчики напишут большую долю комплексных функций пригодности, которые должны выполняться в контексте нескольких точек интеграции для исключения дрейфа сторонних API. Разработчики часто создают между точками интеграции слои защиты от повреждений, чтобы исключить антипаттерн Vendor King, который обсуждается в разделе «Антипаттерн: Vendor King» на с. 201.

Надлежащая связанность

С точки зрения эволюционирования FaaS является привлекательной архитектурой, поскольку она исключает из рассмотрения несколько областей: техническую архитектуру, области эксплуатации и безопасности. Несмотря на то что эта архитектура легко эволюционирует, она имеет серьезные ограничения, связанные с практическими соображениями, и перекладывает большую часть проблем на активирующую функцию. Например, несмотря на то что FaaS будет управлять эластичным масштабированием, вызывающая функция должна управлять любыми транзакциями и сложной координацией. В традиционном приложении координация транзакциями обычно управляется внутренней инфраструктурой. Однако если BaaS не поддерживает это поведение, координация должна переходить на интерфейс пользователя (активирующий сервис).

Разработчики не должны выбирать архитектуру, не оценив ее в отношении реальных проблем, с которыми она столкнется.



Убедитесь, что выбранная архитектура совпадает с проблемной областью. Не пытайтесь приспособить неподходящую архитектуру.

Несмотря на то что бессерверные архитектуры имеют много привлекательных функций, у них есть определенные ограничения. В частности, все решения часто страдают от антипаттерна «Ловушка на последних 10 %» (Last 10% Trap), подробно описанного на с. 206. Большая часть того, что команде необходимо построить, можно сделать легко и быстро, но бывает и так, что разработка полного решения может обмануть их ожидания.

Контроль размера кванта

Размер кванта архитектуры в большой степени определяет, насколько легко будет разработчикам осуществлять эволюционные изменения. Крупные кванты, такие как у монолитной архитектуры и архитектуры SOA на основе событий, с трудом эволюционируют из-за необходимости координации для каждого изменения. Менее связанные архитектуры, такие как архитектуры на основе событий с паттерном брокера и микросервисов, часто намного легче эволюционируют.

Структурные ограничения на эволюционирующую архитектуру зависят от того, насколько хорошо разработчики справились со связанностью и функциональным сцеплением. Эволюционировать легче, если разработчики создали модульную компонентную систему с хорошо определенными точками интеграции. Например, если была создана монолитная архитектура, но с тщательно подобранной модульностью и изоляцией компонентов, то такая архитектура предоставляет больше возможностей для эволюционирования, так как размер архитектурного кванта у нее меньше благодаря связанности.



Чем меньше архитектурный квант, тем более эволюционирующей будет архитектура.

Практический пример: предотвращение циклов компонентов

PenultimateWidgets имеет несколько активно разрабатываемых монолитных приложений. При проектировании компонентов одной из целей архитектуры является создание автономных компонентов — чем больше изолирован код, тем легче с его помощью вносить изменения. Общей проблемой во многих языках с мощной средой разработки является *цикл зависимости пакетов* (package dependency cycle), который описывает общий сценарий, приведенный на рис. 4.14.

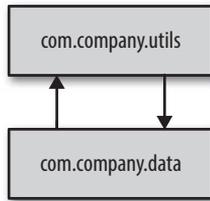


Рис. 4.14. Цикл зависимости пакетов

На рис. 4.14 пакет `com.company.data` импортируется из `com.company.utils`, а `com.company.utils` импортируется из `com.company.data`, при этом ни один из компонентов не может быть использован без вовлечения остальных, что создает цикл компонентов. Очевидно, что циклы нарушают способность изменяться, потому что сложная сеть циклов делает инкрементные изменения затруднительными. Такие языки, как Java или C#, имеют среды разработки, которые помогают разработчикам (с помощью встроенных в среду разработок подсказок), предлагая пропущенный импорт. Поскольку разработчики неявным образом очень много импортируют в процессе программирования с помощью среды разработки, то исключить циклы зависимости пакетов сложно, так как инструментальные средства сопротивляются этим усилиям.

Архитекторы PenultimateWidgets беспокоятся, что разработчики программного обеспечения случайно вставляют циклы между компонентами. К счастью, у них есть механизм для защиты от этих факторов, которые вредят эволюционированию приложений, а именно функции пригодности. Вместо того чтобы отказаться от преимуществ среды разработки, поскольку она способствует появлению плохих привычек, можно с помощью функций пригодности построить сеть безопасности проектирования. Для многих популярных платформ существуют коммерческие инструменты и инструменты с открытым кодом, которые могут помочь распутать циклы. Многие принимают форму инструмента статического анализа кода, который отыскивает циклы, в то время как другие составляют списки задач по рефакторингу кода, чтобы помочь разработчикам это исправить.

Как можно предотвратить появление новых циклов после их удаления? Стандарты программирования не помогают разработчикам в решении этих проблем, так как они помнят бюрократические принципы, лежащие в основе программирования. Вместо этого они предпочитают устанавливать тесты или другие механизмы контроля для защиты от «слишком» полезных инструментов.

Разработчики PenultimateWidgets используют популярный инструмент с открытым исходным кодом для платформы Java JDepend, который включает как текстовые, так и графические интерфейсы для анализа зависимостей. Поскольку JDepend написан на Java, разработчики могут использовать API для написания собственных тестов. Рассмотрим пример 4.1.

Пример 4.1. Использование JDepend для программной идентификации циклов

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class CycleTest extends TestCase {
    private JDepend jdepend;

    protected void setUp() throws IOException {
```

```
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/util/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    /**
     * Проверяет, что ни один пакет не содержит
     * циклов зависимостей пакетов.
     */
    public void testOnePackage() {
        jdepend.analyze();
        JavaPackage p = jdepend.getPackage("com.xyz.thirdpartyjars");
        assertEquals("Cycle exists: " + p.getName(),
            false, p.containsCycle());
    }

    /**
     * Проверяет, что ни один пакет не содержит
     * циклов зависимостей пакетов.
     */
    public void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist",
            false, jdepend.containsCycles());
    }
}
```

В примере 4.1 разработчик добавляет директории, содержащие пакеты для `jdepend`. Затем разработчик может проверить либо одиночный пакет на наличие циклов, либо всю базу кода, как показано в модульном тесте `testAllPackages()`. После того как проект прошел через трудные задачи выявления и удаления циклов, воспользуйтесь модульным тестом `testAllPackages()` вместо функций пригодности для защиты от появления циклов в будущем.

5

Эволюционирующие данные

При участии Прамода Садаладжа

Реляционные и другие типы хранения данных повсеместно используются в проектах по разработке программного обеспечения, представляя форму связи, которая зачастую более проблематична, чем архитектурная связанность. *Данные* являются важной областью, которую необходимо учитывать при разработке эволюционирующей архитектуры. Описание всех аспектов проектирования эволюционирующих баз данных выходит за рамки этой книги. К счастью, наш коллега Прамод Садаладж (Prasad Sadalage) совместно со Скоттом Эмблером (Scott Ambler) написали книгу *Рефакторинг баз данных. Эволюционное проектирование* (<http://datasrefactoring.com/>). Мы включили только отдельные части проектирования баз данных, которые влияют на эволюционирующие архитектуры и призываем наших читателей прочитать эту книгу целиком.

Когда мы говорим *администратор базы данных* (DBA), мы имеем в виду того, кто проектировал структуру данных, писал код доступа к данным и их использования в приложениях, писал код, который выполняет операции с базами данных, сохраняет и улучшает характеристики баз данных и обеспечивает операции резервного копирования и восстановления на случай их утраты. Администраторы и разработчики базы данных часто являются основными строителями приложений, работа которых должна тесно координироваться.

Эволюционное проектирование баз данных

Эволюционное проектирование баз данных возникает, когда разработчики могут строить и развивать структуру базы данных по мере изменений со временем требований. Схемы базы данных являются абстракциями, аналогичными иерархии классов. По мере изменения реального мира эти изменения должны отражаться в абстракциях разработчиков и администратора баз данных. Иначе эти абстракции постепенно утратят синхронизацию с реальным миром.

Эволюционные схемы

Каким образом разработчик архитектуры может построить системы, поддерживающие эволюционное развитие, при этом продолжая использовать традиционный инструмент, такой как реляционные базы данных? Ключ к проектированию эволюционирующей базы данных лежит в эволюционирующих схемах и коде. Непрерывная поставка решает проблему того, как выстроить традиционные разрозненные данные в непрерывный контур обратной связи современных проектов программного обеспечения. Разработчики должны обрабатывать изменения в структуре базы данных так же, как они обрабатывают исходный код: тестировать (tested), версионировать (versioned) и инкрементно изменять (incremental).

Тестирование

Администраторы и разработчики баз данных должны тщательно проверять изменения в схеме базы данных, чтобы гарантировать стабильность. Если разработчики используют инструмент отображения данных, такой как инструмент объектно-ориентированного отображения, они должны учитывать дополнительные функции пригодности, используемые для обеспечения синхронизации отображения со схемами базы данных.

Версионирование

Разработчики и администраторы БД должны вести учет версий схем базы данных вместе с кодом, который она использует. Схемы исходного кода и базы данных являются симбиотическими и не выполняют ни одной функции друг без друга. Инженерные практики, которые искусственно разделяют эти два по необходимости связанных элемента, приводят к потере эффективности.

Инкрементные изменения

Изменения схем базы данных должны появляться по мере того, как вносятся изменения в исходный код: инкрементно, в ходе эволюционирования системы. Современная практика проектирования старается избегать обновления схем базы данных вручную, предпочитая использовать вместо этого автоматизированные инструменты миграции (migration tools).

Инструменты миграции позволяют разработчикам (или администраторам) вносить в базу данных небольшие, инкрементные изменения, которые автоматически применяются как часть конвейера развертывания. Они существуют в широком спектре возможностей от простых инструментальных средств, предназначенных для запуска из командной строки, до таких сложных, как Proton IDE. Если разработчику требуется внести в схему изменение, он пишет небольшой дельта-скрипт, аналогичный приведенному в примере 5.1.

Пример 5.1. 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 кода на языке SQL, аналогичный приведенному в примере 5.1, и автоматически применяет его к экземпляру базы данных разработчика. Если разработчик

спустя определенное время вспоминает, что забыл добавить дату рождения, а не изменить начальную миграцию, он может создать новую миграцию, которая изменит первоначальную структуру, как это показано в примере 5.2.

Пример 5.2. Добавление даты рождения в существующую таблицу с помощью миграции

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

В примере 5.2 разработчик изменил существующую схему, добавив в нее столбец. Некоторые инструменты миграции также могут *отменять* те или иные возможности. Поддержка операции *отмены* дает возможность разработчикам без труда передвигаться вперед и назад в используемых версиях схем базы данных. Предположим, что у проекта версия 101 в репозитории исходного кода и ему нужно вернуться к версии 95. Для исходного кода разработчики просто достают версию 95 из системы управления версиями. Но как они смогут гарантировать то, что схема базы данных подходит для версии 95? Если они используют переходы с возможностью *отмены*, то смогут «отменить» возвращение на более старую версию (95) этой схемы, применяя каждый переход по очереди, чтобы вернуться назад к требуемой версии.

Однако большинство команд отказались от использования *отмены* по трем причинам. Во-первых, если существуют все миграции, то разработчики могут построить базу данных непосредственно для той точки, которая им нужна, не возвращаясь к предыдущим версиям. В нашем примере разработчикам пришлось бы заняться сборкой версий с 1-й по 95-ю, чтобы восстановить версию 95. Во-вторых, зачем поддерживать две правильные версии, как предыдущую, так и предшествующую? Для правильной поддержки *отмены* они должны тестировать код, иногда удваивая нагрузку тестирования. В-третьих, построение полноценной *отмены* иногда приводит к серьезным проблемам. На-

пример, представьте себе, что переход привел к потере таблицы; как теперь скрипт перехода сохранит все данные при операции *отмены*?

После того как разработчики запустили миграции, они считаются неизменными, а изменения моделируются после работы системы двойной записи. Например, предположим, что разработчик Даниэль запустил миграцию из примера 5.2 в качестве 24-й миграции в проекте. Позже он осознал, что `dateofbirth` после этого не нужна. Он мог бы просто удалить 24-ю миграцию, а конечный результат в таблице будет без соответствующего столбца. Однако любой код, написанный в промежутке времени между моментом, когда Даниэль выполнил миграцию и настоящим временем, предполагает наличие столбца с датами рождения и больше не будет работать, если по какой-то причине проекту необходимо вернуться в некоторую промежуточную точку (например, чтобы устранить ошибку). Вместо того чтобы удалять больше не нужный столбец, Даниэль запускает новую миграцию, которая удаляет этот столбец.

Миграции базы данных позволяют администраторам и разработчикам управлять постепенными изменениями схемы и кода, рассматривая каждую из этих частей как единое целое. С помощью включения изменений базы данных в контур обратной связи конвейера развертывания разработчики получают больше возможностей использования автоматизации и облегчения проверок порядка следования операций выполнения проекта.

Интеграция базы данных общего использования

Распространенная схема интеграции, приведенная в этой главе, носит название *интеграции базы данных общего использования* (<http://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html>). Эта схема применяет реляционные базы данных в качестве механизма общего использования данных и представлена на рис. 5.1.

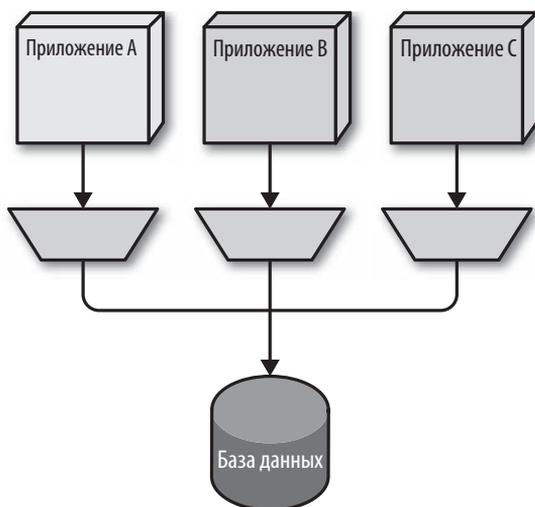


Рис. 5.1. Использование базы данных в качестве точки интеграции

На рис. 5.1 каждое из трех приложений использует одну и ту же реляционную базу данных. Проекты часто по умолчанию используют этот тип интеграции, при этом каждый проект использует одну и ту же реляционную базу данных по причине того, что этого желает руководство. Так почему бы не использовать совместно данные во всех проектах? Разработчики архитектуры быстро обнаружили, что использование базы данных в качестве точки интеграции не дает возможности изменять схему во всех совместно использующих данные проектах.

Что происходит, когда одному из связанных между собой приложений требуется развить возможности с помощью изменения схемы? Если приложение А вносит изменения в схему, это может привести к разрушению двух других приложений. К счастью, как уже отмечалось в вышеупомянутой книге *Рефакторинг баз данных*, для распутывания этого типа связи часто используется паттерн рефакторинга, получивший название *расширения/согласования*. Многие методы рефакторинга баз данных избегают ошибок синхронизации благодаря использованию фазы перехода в рефакторинг, как показано на рис. 5.2.

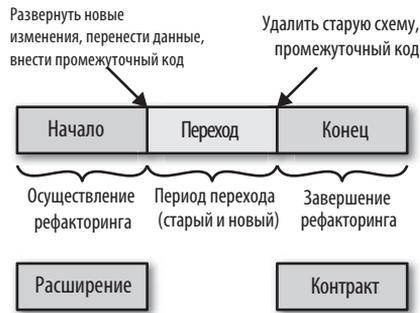


Рис. 5.2. Схема расширения/согласования для рефакторинга базы данных

С использованием этого паттерна, разработчики получили исходное и конечное состояния, сохранив при переходе *старое* и *новое* состояния. Это переходное состояние позволяет использовать возможность вернуться назад, а также предоставляет достаточное время другим системам, чтобы настроиться на это изменение. Для некоторых организаций переходное состояние может продолжаться от нескольких дней до месяцев.

Далее приводится пример *расширения/согласования* в действии. Рассмотрим обычное эволюционное изменение разделения столбца имен на столбец с именем и фамилией, которое компании PenultimateWidgets потребовалось в маркетинговых целях. Для этого изменения у разработчиков есть начальное, расширенное и конечное состояние (рис. 5.3).

На рис. 5.3 полное имя (имя и фамилия) представлено одним столбцом. Во время перехода администратор базы данных компании PenultimateWidgets должен сохранить обе версии, чтобы предотвратить возможное разрушение точек интеграции в базе данных. У администратора есть несколько вариантов того, как выполнить разделение столбца с полным именем на столбец с именем и столбец с фамилией.

1-й вариант: никаких точек интеграции, никаких унаследованных данных

В этом случае у разработчиков нет никаких других систем и никаких существующих данных для управления, поэтому они могут добавить новые столбцы и удалить старый столбец, как показано в примере 5.3.

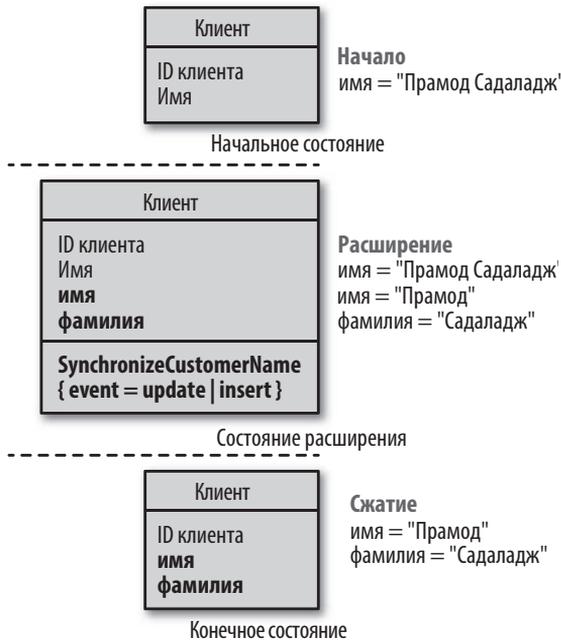


Рис. 5.3. Три состояния рефакторинга расширения/согласования

Пример 5.3. 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 *1-го варианта* рефакторинг выполняется напрямую, при этом администратор вносит необходимые изменения и забывает об этом.

2-й вариант: данные наследуются, но нет никаких точек интеграции

В этом сценарии разработчики берут существующие данные для их миграции в новые столбцы, но при этом у них нет никаких внешних систем, о которых пришлось бы беспокоиться. Для извлечения необходимой информации из существующего столбца им необходимо

создать функцию, которая будет управлять переносом данных, как показано в примере 5.4.

Пример 5.4. Наследуемые данные, но 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;
```

В этом сценарии администратор базы данных должен извлечь и осуществить миграцию данных, однако это нельзя считать операцией, выполненной напрямую.

3-й вариант: существующие данные и точки интеграции

Это самый сложный и, к сожалению, наиболее распространенный сценарий. Компании необходимо выполнить миграцию имеющихся данных в новые столбцы, в то время как внешние системы зависят от столбца имя, который их разработчики не могут перенести, чтобы использовать новые столбцы в требуемых временных рамках. Требуемый SQL фрагмент появляется в примере 5.5.

Пример 5.5. 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;
```

Для создания переходного состояния (см. пример 5.5) администратор базы данных добавляет в базу данных триггер, который перемещает данные из столбца с **полным именем** в новый столбец с **именем** и столбец с **фамилией**, когда остальные системы вставляют данные в эту базу данных, предоставляя новой системе доступ к тем же самым данным. Аналогичным образом разработчики или администратор базы данных объединяют **имя** и **фамилию** в столбце с **полным именем**, когда новая система вставляет данные так, что остальные системы имеют доступ к своим надлежащим образом форматированным данным.

После того как остальные системы изменяют свой доступ, чтобы пользоваться новой структурой (с отдельными именами и фамилиями), может быть выполнена фаза *сжатия* и старый столбец удаляется:

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

Если существует множество данных и удаление столбца займет много времени, администратор базы данных может установить для столбца статус «не используется» (если база данных поддерживает эту функцию):

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

После исключения столбца со старыми данными, если нужна версия только для чтения предыдущей схемы, администратор базы данных может добавить функциональный столбец, чтобы сохранить доступ для чтения в базу данных:

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

Как было показано в каждом сценарии, администратор и разработчики базы данных могут использовать оригинальные функции базы данных, чтобы построить эволюционирующие системы.

Расширение/согласование является подсистемой паттерна параллельного изменения (<https://martinfowler.com/bliki/ParallelChange.html>), который широко используется схемой для безопасного применения, исключая возвращение назад изменения интерфейса.

Ненадлежащая связанность данных

Данные и базы данных образуют составную часть большинства современных архитектур программного обеспечения, и игнорирующие этот ключевой аспект разработчики испытывают трудности при попытке эволюционирования своей архитектуры.

Базы данных и их администраторы создают определенную проблему во многих организациях, потому что, неважно по какой именно причине, их инструменты и опыт проектирования являются устаревшими по сравнению с традиционно развивающимся миром. Например, ежедневно используемые администратором инструменты чрезвычайно примитивны по сравнению с любой интегрированной средой разработки. Функции, которыми широко пользуются разработчики, не существуют для администраторов баз данных: поддержка рефакторинга, внеконтейнерное тестирование, модульные тесты, имитация (mocking), функции «заглушки» (stubbing) и т. п.

АДМИНИСТРАТОРЫ БАЗ ДАННЫХ, ПОСТАВЩИКИ И ВЫБОР ИНСТРУМЕНТОВ

Почему мир данных так сильно отстает от мира практики проектирования программного обеспечения? Администраторам требуется использовать многое из того, что необходимо разработчикам: тестирование, рефакторинг и т. п. При том, что инструмент разработок продолжает развиваться, этот уровень инноваций не проникает в мир данных. Это происходит не потому, что необходимые инструменты недоступны: существует несколько инструментов сторонних производителей, способных обеспечить улучшенную поддержку, но они плохо продаются. Почему?

Поставщики баз данных создали интересные взаимоотношения между собой и своими клиентами. Например, администратор баз

данных (DBA) для поставщика баз данных *X* имеет иррациональный уровень приверженности этому поставщику, поскольку следующая работа у DBA появится частично из-за того, что компания является сертифицированным администратором баз данных поставщика *X*, а не из-за имеющейся у них работы. Поэтому поставщики баз данных имеют секретные армии внутри предприятий во всем мире, в которых понятие преданности относится к поставщику, а не к компании. DBA в этой ситуации игнорируют инструменты и другие средства разработки, которые не приходят от поставщика. Результатом является застой на уровне инноваций для практик проектирования.

DBA рассматривают своих поставщиков баз данных как источник всего тепла и света во Вселенной, и им все равно, что исходит от другой темной материи. Неблагоприятным побочным эффектом этого явления является застой в развитии инструментов баз данных по сравнению с инструментами разработчиков. В результате этого степень несогласованности между разработчиками и администраторами баз данных растет еще больше, поскольку они не используют совместно одни и те же методы проектирования. Убеждение администраторов в необходимости принять практику непрерывной поставки заставляет их использовать новый инструмент, дистанцируясь тем самым от поставщика баз данных, чего они стараются избежать.

К счастью, популярность средств открытого доступа и баз данных NoSQL привела к началу уничтожения гегемонии поставщиков баз данных.

Двухфазная фиксация транзакций

Когда разработчики архитектуры обсуждают связанность, это обсуждение обычно вращается вокруг классов, библиотек и прочих аспектов технической архитектуры. Однако в большинстве проектов существуют и другие средства связи, включая транзакции.

Транзакции являются специальной формой связанности, так как поведение при транзакции не проявляется в традиционных архитекту-

роцентричных инструментах. Разработчики архитектуры могут без труда определить афферентные и эфферентные соединения классов с помощью разнообразных инструментов. У них были значительно более тяжелые времена для определения охвата контекста транзакций. Как связь между схемами препятствует эволюционированию, так же и транзакции связывают определенным образом составляющие части, затрудняя эволюционирование.

Транзакции появились в бизнес-системах по разным причинам. Во-первых, специалистам по анализу нравится идея транзакций как операция, которая на короткое время *останавливает все вокруг* ради определенного контекста, независимо от возникающих при этом технических проблем. Глобальная координация в сложных системах труднореализуема, а транзакции представляют одну из ее форм. Во-вторых, границы транзакций часто говорят о том, как при их реализации в действительности связаны между собой бизнес-концепции. В-третьих, администраторы баз данных могут обладать контекстами транзакций, затрудняя разделение данных на части, чтобы повторить связанность, обнаруженную в технической архитектуре.

Разработчики рассматривают транзакции как точки связи при попытке перевести тяжелые системы транзакций в соответствующие архитектурные шаблоны, такие как микросервисы, которые накладывают тяжело разделяемые нагрузки. Сервис-ориентированные архитектуры со значительно менее строгими границами сервисов и требованиями к разделению данных лучше подходят для системы транзакций. Мы обсудили различия между этими типами архитектур в разделе «Сервис-ориентированные архитектуры» на с. 113.

В главах 1 и 4 мы обсуждали определение границ архитектурного кванта, наименьшего развертываемого модуля архитектуры, который отличается от традиционного представления о сцеплении благодаря включению зависимых компонентов, таких как базы данных. Связанность, создаваемая базами данных, более значительная, чем традиционная связанность из-за границ транзакций, которые часто определяют то, как работают бизнес-процессы. Разработчики архитектур иногда ошибаются, пытаясь построить архитектуру с меньшей

степенью детализации, чем принято считать естественным для соответствующей предметной области. Например, архитектуры микросервиса не очень хорошо подходят для сильно транзакционных систем, поэтому квант целевого сервиса очень маленький. Сервис-ориентированные архитектуры работают лучше из-за менее строгих требований к размеру кванта.

Разработчики должны учитывать все характеристики связанности своей прикладной области: классы, пакеты/область имен, библиотека и платформа, схемы данных и контексты транзакций. Игнорирование любой из этих областей (или их взаимодействий) создает проблемы при попытке эволюционного развития архитектуры. В физике *ядерные силы сильного взаимодействия*, которые связывают атомы вместе, являются одними из самых сильных воздействий из известных на сегодняшний день. Контексты транзакций действуют для архитектурных квантов как ядерные силы.



Транзакции базы данных действуют как силы ядерного взаимодействия, связывая кванты воедино.

Несмотря на то что системы часто не могут избежать транзакций, разработчики должны по возможности пытаться ограничить контексты транзакций, потому что они формируют плотный узел соединений, снижая способность изменять компоненты или сервисы, не влияя на остальные элементы системы. Важнее то, что разработчики, планируя архитектурные изменения, должны учитывать такие характеристики транзакций, как их границы.

Как упоминается в главе 8, при переходе с монолитной архитектуры на архитектуру с большим уровнем детализации начинать следует с меньшего количества более крупных сервисов. При построении архитектуры микросервисов с нуля разработчики должны быть внимательны при ограничении размера сервиса и контекстов данных. Но не следует воспринимать название *микросервисов* слишком буквально, каждый сервис не должен быть маленьким, он должен быть способным захватывать полезный ограниченный контекст.

При реконструкции схемы существующей базы часто сложно добиться надлежащей степени гранулярности. Многие администраторы баз данных предприятий тратят годы, соединяя схемы баз данных вместе, и при этом они не заинтересованы в выполнении обратной операции. Часто необходимый контекст транзакции для поддержки предметной области определяет наименьшую гранулярность, которую разработчики могут использовать в сервисах. Несмотря на то что разработчики могут стремиться создать меньший уровень детализации, их усилия приводят к ненадлежащей связанности, если эти попытки создают рассогласование с данными. Разработка архитектуры, структурно конфликтующей с проблемой, которую пытаются решить разработчики, представляет собой разрушительную версию метапрограммирования, описанную в разделе «Миграция архитектур» на с. 165.

Возраст и качество данных

Другим нарушением функциональности, проявляющимся в крупных компаниях, является фетишизация данных и баз данных. Мы слышали, что говорил один руководитель технического отдела: «Я действительно не обращаю внимания на то, как работают приложения, поскольку у них короткий срок службы, *но мои схемы данных ценны, потому что они живут вечно!*» Несмотря на то что схемы действительно меняются реже, чем код, схемы базы данных все же представляют собой абстракцию реального мира. Несмотря на вызванное этим неудобство реальный мир имеет привычку изменяться со временем. Те администраторы баз данных, которые уверены, что схемы никогда не меняются, игнорируют реальность.

Но если администраторы никогда не перестраивают базу данных, чтобы заставить измениться схемы, тогда как же их заставить измениться, чтобы принять новые абстракции? К сожалению, *добавление другой промежуточной таблицы* является распространенным процессом, который применяет администратор, чтобы расширить определение схем. Вместо того чтобы заставить схему измениться и подвергнуть существующие системы риску разрушения, они добавляют новую таблицу,

соединяя ее с исходной, используя для этого примитивы реляционной базы данных. Несмотря на то что эти работы — краткосрочные, они нарушают лежащие в основе реальные абстракции: в реальном мире один объект представлен множеством характеристик. Спустя время администратор, редко реструктуризирующий схемы, создает все более окаменелый мир с коварным группированием и стратегией связанности. Если администраторы не реструктуризуют базу данных, то они вместо сохранения ресурсов предприятия создают наложенные друг на друга остатки предыдущих версий схемы, используя для этого промежуточные таблицы.

Качество наследуемых данных представляет другую большую проблему. Часто данным удается пережить многие поколения программного обеспечения, каждое со своей собственной системой хранения данных, что приводит к тому, что данные в лучшем случае оказываются несогласованными, а в худшем — непригодными к использованию. Стремление разными путями удерживать каждый фрагмент данных связывает архитектуру с прошлым, вынуждая придумывать обходные пути, чтобы все работало успешнее.

Прежде чем приступать к разработке эволюционирующей архитектуры, следует убедиться, что разработчики могут так же эволюционировать данные — и в отношении схемы, и в отношении качества. Плохая структура данных требует рефакторинга, и администраторы баз данных должны выполнить любые необходимые действия для поддержания качества данных. Мы предпочитаем решать эти проблемы на ранней стадии, а не разрабатывать изменения постоянно протекающих механизмов для решения этих проблем.

Предыдущие схемы и унаследованные данные представляют определенную ценность, но они также накладывают издержки на возможность эволюционировать. Разработчики, администраторы баз данных и представители бизнеса должны иметь возможность откровенно обсудить то, что представляет *ценность* для организации — сохранять унаследованные данные или иметь возможность эволюционных изменений. Необходимо рассмотреть действительно представляющие ценность данные и сохранить их, а для более старых данных — со-

хранить возможность на них ссылаться, но вывести их из магистрали эволюционного развития.



Отказ от рефакторинга схем или удаление старых данных связывает вашу архитектуру с прошлым, что затрудняет ее рефакторинг.

Практический пример: эволюционирование методов маршрутизации в PenultimateWidgets

Компания PenultimateWidgets решила внедрить новую схему маршрутизации между страницами, предоставив пользователям навигационную цепочку. Это означало, что было выполнено изменение способа прокладки маршрута между страницами (используя внутреннюю платформу). Страницы, использующие новый механизм построения маршрута, требуют больше контекста (исходная страница, состояние рабочего потока и т. п.) и поэтому — больше данных.

Внутри кванта сервиса маршрутизации у PenultimateWidgets сейчас есть единая таблица для управления маршрутами. Для новой версии разработчикам требуется больше информации, поэтому структура таблицы будет сложнее. Рассмотрим начальную точку, представленную на рис. 5.4.



Рис. 5.4. Начальная точка для использования новой маршрутизации

В новом сервисе маршрутизации PenultimateWidgets не все страницы будут использоваться одновременно, потому что различные бизнес-

модули работают с разной скоростью. Таким образом, сервис маршрутизации должен поддерживать старую и новую версии. В главе 6 описано, как это можно реализовать с помощью маршрутизации. В этом случае мы должны использовать на уровне данных один и тот же сценарий.

С помощью схемы расширения/отладки разработчик может создать новую структуры сервиса маршрутизации и с помощью вызова сервиса сделать ее доступной. Внутри обе таблицы маршрутизации имеют триггер, связанный со столбцом маршрутов, так что изменения в одной таблице автоматически копируются в другую, как это показано на рис. 5.5.

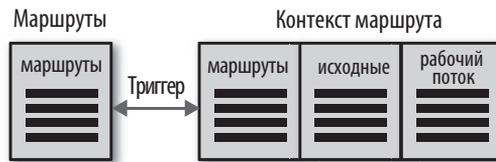


Рис. 5.5. Переходное состояние, в котором сервис поддерживает обе версии маршрутизации

Как можно видеть на рис. 5.5, этот сервис может поддерживать оба API, пока разработчикам нужен старый сервис маршрутизации. По сути, это приложение теперь поддерживает две версии информации маршрутизации.

Когда старый сервис больше не нужен, разработчики сервиса маршрутизации могут удалить старую таблицу и триггер, как это показано на рис. 5.6.



Рис. 5.6. Конечное состояние таблиц с маршрутами

На рис. 5.6 все сервисы перешли на использование новых возможностей маршрутизации, позволив удалить старый сервис. Это соответствует рабочему потоку, показанному на рис. 5.2.

Эта база данных может эволюционировать вместе с архитектурой, пока разработчики используют надлежащую практику проектирования, такую как непрерывная интеграция, управление версиями исходного кода и т. п. Возможность легко изменять схему базы данных является решающей: база данных представляет собой абстракцию, основанную на реальном мире, которая может неожиданно измениться. Несмотря на то что абстракции данных сопротивляются изменениям лучше, чем их характеристики, они все еще должны эволюционировать. При построении эволюционирующей архитектуры разработчики должны рассматривать данные как основную проблему.

Рефакторинг баз данных является важным навыком и умением администратора и разработчика БД, который необходимо оттачивать. Данные — это фундаментальное понятие для многих приложений. Чтобы построить эволюционирующие системы разработчики и администраторы должны выбрать эффективную практику обращения с данными вместе с другими современными практиками проектирования.

6

Построение архитектуры с эволюционным развитием

До сих пор мы рассматривали отдельно три основных аспекта эволюционирующей архитектуры, а именно функции пригодности, инкрементное изменение и надлежащую связанность. У нас не было достаточно контекста, чтобы связать их вместе.

Многие концепции, которые мы обсуждаем в этой книге, не являются новыми идеями, но на них интересно будет взглянуть по-новому. Например, тестирование существовало многие годы, но без функций пригодности, делающих акцент на проверке архитектуры. *Непрерывная поставка* определяет идею конвейеров развертывания. Эволюционирующая архитектура показывает разработчикам архитектуры реальную пользу этой способности.

Многие организации используют практику непрерывной поставки как способ повысить эффективность проектирования при разработке программного обеспечения, что само по себе заслуживает внимания. Однако мы делаем следующий шаг, используя эти возможности, чтобы создать что-то более сложное, а именно архитектуру, которая эволюционирует вместе с реальным миром.

Итак, как разработчики могут использовать преимущества этих методов в проектах, как существующих, так и новых?

Техники

Разработчики архитектуры могут задействовать эти методы для построения эволюционирующей архитектуры в три этапа.

1. Определить области, затрагиваемые эволюционным развитием

Прежде всего, разработчик должен определить, какие области архитектуры ему хотелось бы защитить по мере эволюционирования архитектуры. К ним всегда относится техническая архитектура и часто проектирование структуры данных, безопасность, масштабируемость и прочие области, которые разработчик считает важными. Это также должно включать другие заинтересованные группы организации, например группу предметной области, операций, безопасности и т. п. *Обратный закон Конвея* (описанный в разделе «Закон Конвея» на с. 33) может быть полезным, потому что он поощряет многофункциональные команды. В основном это является распространенным поведением разработчиков архитектуры в самом начале работы над проектом, когда выявляются характеристики архитектуры, которые разработчики хотят поддерживать.

2. Определить для каждой области функцию(-и) пригодности

В одной области часто содержится несколько функций пригодности. Например, разработчики связывают набор показателей качества кода в конвейер развертывания, чтобы обеспечить характеристики исходного кода и исключить циклы зависимости компонентов. Разработчики документируют решения о том, какие из областей заслуживают в настоящее время внимания в простом формате, таком как вики-формат. Затем для каждой области они принимают решение, какие части могут при эволюционировании системы проявлять нежелательное поведение, и в итоге определяют на основе этого функции пригодности. Функции пригодности могут быть автоматизированными или запускаемыми вручную и в некоторых случаях могут быть необходимы.

3. Использовать конвейер развертывания для автоматизации функций пригодности

И наконец, разработчики должны оказывать поддержку инкрементным изменениям в проекте, определяя этапы в конвейере развертывания для применения функций пригодности, и управлять процедурой развертывания, включающей инициализацию машины, тестирование и другие операции DevOps. Инкрементное изменение является двигателем эволюционирующей архитектуры, позволяя активно проверять функции пригодности с помощью конвейера развертывания и допуская высокую степень автоматизации, которая делает многие простые задания, такие как развертывание, невидимыми. Время цикла является мерой эффективности проектирования непрерывной поставки. В ответственность разработчиков проектов, поддерживающих эволюционирующие архитектуры, входит также поддержание приемлемого времени цикла. Время цикла является важной характеристикой инкрементного изменения, потому что многие другие показатели являются производными времени цикла. Например, скорость появления в архитектуре новых поколений пропорциональна времени ее цикла. Другими словами, время цикла проекта удлиняется, это тормозит скорость появления новых поколений, что оказывает влияние на эволюционирование.

Несмотря на то что выявление важных областей и определение функций пригодности происходит в начале нового проекта, это также определяет текущую активность для нового и существующего проекта. Программное обеспечение страдает от проблем *неизвестных неизвестных*: разработчики не могут предвидеть все. При построении некоторые части архитектуры часто показывают тревожные признаки, и разработка функций пригодности может предотвратить развитие этого функционального нарушения. Несмотря на то что некоторые функции пригодности естественным образом появляются в самом начале проекта, множество других не проявят себя до появления в архитектуре точки напряжения. Разработчики архитектуры должны внимательно следить за ситуациями, в которых нарушаются нефункциональные требования, и добавлять в архитектуру новые функции пригодности для исключения последующих проблем.

Проекты с нуля

Разработка эволюционирования в новых проектах намного легче, чем модернизация уже существующих. Во-первых, у разработчиков есть возможность сразу использовать постепенные изменения, создавая конвейер развертывания в самом начале разработки проекта. Функции пригодности легче выявить и распланировать до появления кода, что облегчит размещение сложных функций пригодности, так как автоматическая кодогенерация проекта существует с самого начала проекта. Во-вторых, разработчикам архитектуры не требуется распутывать любые нежелательные точки связанности, которые незаметно проникают в существующие проекты. Разработчик архитектуры может также установить на место все количественные метрики и прочие средства проверки для обеспечения целостности архитектуры по мере изменений проекта.

Разработка новых проектов, которые справлялись бы с неожиданными изменениями, намного легче, если разработчик выбирает правильную схему архитектуры и практику проектирования, способствующие эволюционированию архитектуры. Например, архитектуры микросервисов предоставляют чрезвычайно низкий уровень связанности и высокую степень постепенности изменений (помимо других факторов, способствующих популярности микросервисов), что делает этот тип архитектуры удачным вариантом.

Настройка существующих архитектур

Возможность добавления способности к эволюционированию в существующие архитектуры зависит от трех факторов: связанности компонентов, практики проектирования и способности разработчика подбирать функции пригодности.

Надлежащие связанность и сцепление

Связанность компонентов в значительной степени определяется способностью к эволюционированию технической архитектуры.

И все же наилучшая с точки зрения эволюционирования техническая архитектура обречена, если используется жесткая схема данных. Четко несвязанные системы облегчают эволюционирование; узлы многочисленных соединений вредят ей. Чтобы построить истинно эволюционирующую систему, разработчики должны учитывать все затронутые области архитектуры.

За рамками технических аспектов связанности разработчики также должны учитывать и защищать функциональное сцепление компонентов этой системы. При переходе из одной архитектуры в другую функциональное сцепление определяет конечную грануляцию реконструированных компонентов. Это не означает, что разработчики не могут разложить компоненты до смехотворного уровня, это скорее значит, что компоненты должны иметь надлежащий размер, который основывается на контексте проблемы. Например, некоторые бизнес-проблемы более тесно связаны друг с другом, например, в случае транзакционных систем. Попытка построить сильно несвязанную архитектуру, которая противоречит проблеме, непродуктивна.



Понять бизнес-проблемы перед выбором архитектуры.

Несмотря на то что этот совет кажется очевидным, мы постоянно сталкиваемся с командами, которые страдают от того, что выбрали блестящий новый паттерн архитектуры, а не наиболее подходящий. Частично выбор архитектуры обусловлен пониманием того, где согласуется проблема с технической архитектурой.

Практики проектирования

Практики проектирования играют роль при определении того, насколько эволюционирующей может быть архитектура. Несмотря на то что практика непрерывной поставки не гарантирует эволюционирования архитектуры, без нее обойтись практически невозможно.

Многие команды для обеспечения эффективности принимались улучшать практики проектирования. Однако как только эти практики прочно устанавливаются, они становятся строительными блоками для таких расширенных возможностей, как эволюционная архитектура. Поэтому способность построить эволюционирующую архитектуру является стимулом для повышения эффективности.

Многие компании так и застряли в переходной области между старыми и новыми практиками проектирования. Они могут достать висящие низко плоды, такие как непрерывная интеграция, но у них остается слишком много выполняемых вручную тестирований. Несмотря на то что это удлиняет время цикла, выполняемые вручную этапы важно включить в конвейер развертывания. Во-первых, это позволяет одинаково рассматривать каждый этап строительства приложения как этап в конвейере. Во-вторых, по мере того как команды медленно автоматизируют больше частей развертывания, выполняемые вручную этапы могут стать автоматизированными безо всяких нарушений. В-третьих, изучение каждого этапа дает сведения о механических частях сборки, улучшая обратную связь и поощряя усовершенствования.

Самым большим общим препятствием для создания эволюционной архитектуры являются трудноразрешимые операции. Если разработчикам не удастся легко развернуть изменения, все части цикла обратной связи затрудняются.

Функции пригодности

Функции пригодности формируют защитную подложку эволюционирующей архитектуры. Если разработчики проектируют систему с определенными характеристиками, эти особенности могут вступать в противоречие со способностью к тестированию. К счастью, современные практики проектирования значительно улучшили тестируемость, сделав возможным автоматически проверять вызывающие ранее затруднения характеристики архитектуры. Эта область эволюционирующей архитектуры требует основного объема работ, но функции

пригодности позволяют одинаково эффективно решать ранее не сопоставимые проблемы.

Мы призываем архитекторов начать думать обо всех видах механизмов проверки архитектуры как о функциях пригодности, в том числе о тех вещах, которые они ранее рассматривали по ситуации. Например, во многих архитектурах есть соглашение об уровне обслуживания по масштабируемости и соответствующим тестам. У них также есть правила, касающиеся требований безопасности, с сопровождающими механизмами проверки. Разработчики часто воспринимают эти категории как не связанные между собой, но они обе нацелены на одно и то же: проверить некоторые особенности архитектуры. При рассмотрении всех архитектурных проверок как функций пригодности появляется больше согласованности, когда определяются автоматизация и другие полезные взаимодействия.

РЕФАКТОРИНГ ИЛИ РЕСТРУКТУРИЗАЦИЯ

Разработчики иногда берут термины, которые классно звучат, и используют их в более широком смысле, как, например, *рефакторинг*. Согласно определению Мартина Фаулера, *рефакторинг* — это изменение во внутренней структуре программного кода, без изменения наблюдаемого поведения. Для многих разработчиков *рефакторинг* стал синонимом *изменения*, но между ними существуют принципиальные различия.

Крайне редко команда производит рефакторинг архитектуры; наоборот, они *реструктурируют* ее, внося существенные изменения как в ее структуру, так и в поведение. Паттерны архитектуры используются частично, чтобы сделать определенные характеристики архитектуры в том или ином приложении основными. Переход из одного паттерна в другой приводит к изменению приоритетов, что не является рефакторингом. Например, разработчики могут выбрать для обеспечения масштабируемости архитектуру, основанную на событиях. Если команда переходит на другой паттерн архитектуры, то она может не поддерживать тот же уровень масштабируемости.

Применение коммерческой продукции

Во многих организациях разработчики не владеют всеми частями, составляющими экосистему. В крупных компаниях преобладают готовые решения и пакеты программного обеспечения с применением коммерческого ПО (COTS — commercial off-the-shelf), что создает проблемы для разработчиков эволюционирующих систем.

Системы COTS должны эволюционировать вместе с остальными приложениями, используемыми предприятием. К сожалению, эти системы не поддерживают надлежащим образом эволюционное развитие.

Инкрементное изменение

Большинство коммерческого программного обеспечения значительно отстает от отраслевых стандартов автоматизации и тестирования. Разработчики архитектур и программного обеспечения должны зачастую ограждать точки интеграции и разрабатывать то, что можно протестировать, часто рассматривая всю систему как черный ящик. Повышение динамичности с помощью конвейеров развертывания, DevOps и других современных практик создает проблемы для разработчиков.

Надлежащая связанность

Пакет программного обеспечения часто приводит к самым тяжелым проблемам в отношении связанности. Вообще, эта система непрозрачна при использовании разработчиками для интеграции API. К сожалению, API страдает от проблем, описанных в разделе «Антипаттерн: ловушка на последних 10 %» на с. 206, предоставляя при этом разработчикам достаточную степень универсальности, чтобы завершить полезную работу.

Функции пригодности

Добавление в пакет программного обеспечения функций пригодности является, пожалуй, самым большим препятствием для развития. Как правило, инструменты этого типа не обнажают достаточное число внутренних элементов, чтобы дать возмож-

ность выполнить модульное тестирование или тестирование компонентов, делая тестирование поведенческой интеграции последней надеждой. Такие тесты менее желательны, потому что они, имея неизбежно менее детализированную структуру, должны выполняться в сложной среде и должны тестировать большую часть поведения системы.



Необходима тщательная работа по удержанию точек интеграции на уровне развития проекта. Если это невозможно, следует понимать, что для некоторых частей системы разработчику будет легче обеспечить способность к эволюционированию, чем для остальных.

Другой вызывающей беспокойство точкой связанности, вносимой некоторыми поставщиками пакетов программного обеспечения, являются непрозрачные экосистемы баз данных. В лучшем случае пакет программного обеспечения полностью управляет состоянием базы данных, раскрывая представляющие интерес значения посредством точек интеграции. В худшем случае база данных поставщика *сама* является точкой интеграции для остальной системы, сильно усложняя возможность проводить изменения с каждой из сторон API. В этом случае разработчики и администраторы базы данных должны бороться за управление базой данных вне пакета программного обеспечения ради хоть какой-то надежды на эволюционирование.

Если у вас есть необходимый пакет программного обеспечения, архитекторы должны создать максимально надежный набор функций пригодности и автоматизировать их работу при любой возможности. Отсутствие доступа к внутренним элементам вызывает проведение тестирования с помощью менее желательных методов.

Миграция архитектур

Многие компании прекратили миграцию архитектур из одного типа в другой. Например, разработчики архитектуры выбирают в самом начале ИТ-деятельности компании простую для понимания схему

архитектуры, часто слоистую монолитную архитектуру. По мере развития компании разработчики начинают подвергаться воздействию нагрузок. Один из наиболее распространенных путей миграции — это переход из монолитной в какую-либо из разновидностей сервис-ориентированных архитектур. Он возникает по причине предметно-центрированного сдвига в архитектурном мышлении, рассмотренного в разделе «Микросервисы» на с. 118. Многих разработчиков привлекает архитектура микросервисов с высоким уровнем эволюционирования в качестве цели миграции, но часто это достаточно сложно осуществить, в основном из-за существующей связанности.

Когда разработчики планируют миграцию архитектуры, они обычно учитывают характеристики связанности классов и компонентов, но при этом игнорируют многие другие области, затрагиваемые эволюционированием данных. Связанность по транзакциям так же реальна, как связанность между классами, и столь же коварна для устранения при реструктуризации архитектуры. Эти точки внеклассовой связанности оказывают большое сопротивление при попытке разбить существующие модули на мелкие составляющие.

Многие старшие разработчики год за годом создают одни и те же приложения и скучают от монотонности. Большинство разработчиков скорее *напишут* фреймворк, чем будут *использовать* существующий для создания чего-нибудь полезного: *метапрограммирование интереснее, чем простая работа*. Работа представляет собой скучное, прозаичное и однообразное занятие, в то время как разработка чего-то нового захватывает.

Это проявляется двояко. Во-первых, многие старшие разработчики стараются написать инфраструктуру, которую используют (часто с открытым исходным кодом) другие разработчики вместо того, чтобы использовать существующее программное обеспечение. Однажды нам пришлось работать с клиентом, который находился на переднем крае технологий. Они построили собственный сервер приложений, веб-фреймворк на Java и почти что все остальные биты инфраструктуры. В какой-то момент мы спросили, построили ли они или нет собствен-

ную операционную систему, и они ответили, что нет. Мы спросили: «Почему нет?! Вы построили все с нуля!»

Если хорошо подумать, компании нужны были возможности, которые были недоступны. Однако когда становятся доступны инструменты с открытым кодом, у них уже есть собственная, с любовью построенная инфраструктура. Вместо того чтобы перейти на более стандартный стек, они предпочли сохранить собственный из-за незначительных различий в подходе. Спустя десять лет их лучшие разработчики постоянно работали в режиме обслуживания, устраняя ошибки собственного сервера, добавляя функции в веб-фреймворк и выполняя другие рутинные операции. Вместо создания инноваций для построения улучшенных приложений они постоянно тянули ляжку, строя связанное программное обеспечение.

Архитекторы не защищены от синдрома «метапрограммирование интереснее обычной работы». Это проявляется в выборе ненадлежащих, но вызывающих повышенный интерес типов архитектуры, таких как микросервисы.



Не стройте архитектуру просто потому, что это забавное метапрограммирование.

Шаги миграции

Многие архитектуры оказываются перед проблемой миграции обновленного монолитного приложения на более современный метод, основанный на сервисе. Опытные разработчики понимают, что в приложении существует масса точек связанности, и одна из первых задач при распутывании базы кода состоит в том, чтобы понять, как все соединено между собой. При составлении монолитной архитектуры разработчик должен учитывать связанность и сцепление, чтобы найти подходящий баланс. Например, одним из наиболее строгих ограничений в архитектуре микросервисов является ограничение времени, в течение которого база данных находится внутри ограниченного контекста сервиса. При декомпозиции монолитной архитектуры,

даже если можно разбить классы на достаточно небольшие части, декомпозиция контекстов транзакций на более мелкие части может оказаться непреодолимой преградой.



При реконструировании архитектуры следует учитывать *все* области архитектуры.

Многие разработчики остановились на миграции монолитных приложений в сервис-ориентированную архитектуру. Рассмотрим начальную архитектуру, приведенную на рис. 6.1.

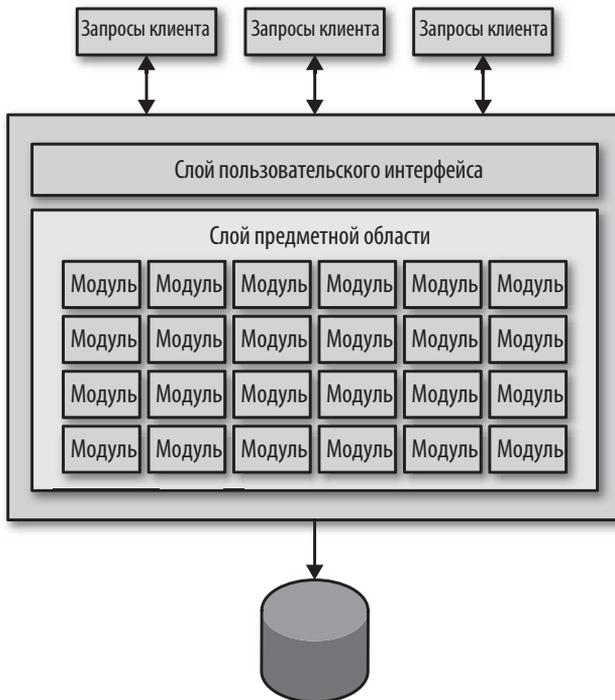


Рис. 6.1. Монолитная архитектура в качестве начального варианта для миграции на архитектуру «все общее»

Разработка приложения с чрезмерной гранулярностью легче реализуется в новых проектах, но затруднительна в существующих миграциях.

Поэтому как мы могли бы осуществить миграцию из представленной на рис. 6.1 архитектуры в сервис-ориентированную архитектуру, показанную на рис. 6.2?

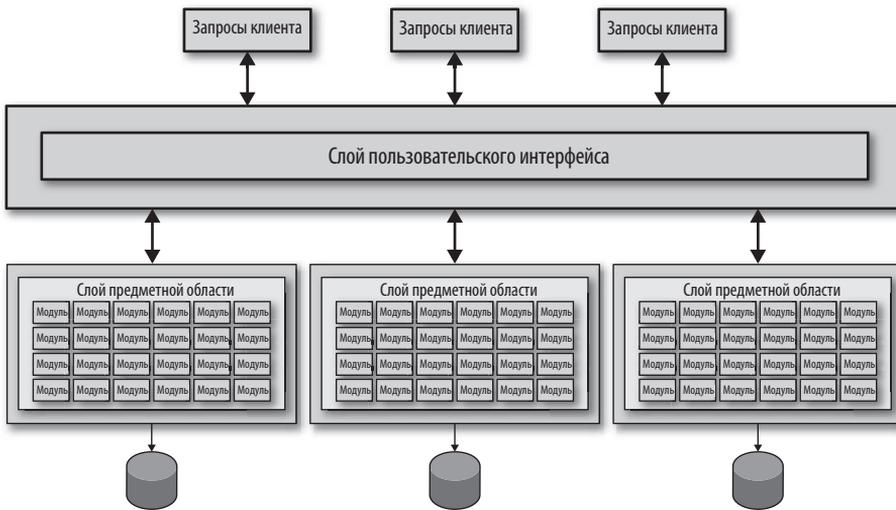


Рис. 6.2. Результат перехода в сервис-ориентированную архитектуру, архитектура «как можно меньше общего»

Выполнение представленной на рис. 6.1 и 6.2 миграции происходит с массой проблем: степень гранулярности сервисов, границы транзакций, проблемы с базой данных и с тем, как пользоваться совместными библиотеками. Разработчики должны понимать, для чего они хотят выполнить миграцию, и это должна быть более серьезная причина, чем «это современная тенденция». Декомпозиция архитектуры на области, наряду с улучшенным составом команды и операционной развязкой, позволяет легче реализовать инкрементные изменения, которые являются одним из строительных блоков эволюционирующей архитектуры, потому что акцент делается на артефактах физической работы.

При декомпозиции архитектуры ключевым фактором является подбор гранулярности сервисов. Создание крупных сервисов облегчает решение таких проблем, как контексты транзакций и управление ими, но мало что дает для разделения монолитной архитектуры на

небольшие составляющие. Слишком мелкие компоненты приводят к повышенной оркестровке, избыточности связи и взаимозависимости компонентов.

Для первого шага в миграции архитектуры разработчики определяют новые границы сервисов. Команды могут принять решение разбить монолитную архитектуру на сервисы, используя различные разделения на составляющие.

Группы предметной области

Предметная область может иметь четкие составляющие, которые напрямую отображают IT-возможности. Построение программного обеспечения, которое имитирует существующую иерархию связи в предметной области, определенно подпадает под действие закона Конвея (см. раздел «Закон Конвея» на с. 33).

Границы транзакций

Многие предметные области имеют протяженные границы транзакций, которых они должны придерживаться. При декомпозиции монолитной архитектуры разработчики часто обнаруживают, что связанность по транзакциям тяжелее всего разбить на части, как это описано в разделе «Двухфазная фиксация транзакций» на с. 149.

Цели развертывания

Инкрементное изменение позволяет разработчикам избирательно выпускать код по разным графикам. Например, отделу маркетинга может потребоваться больший интервал обновлений, чем складам. Разделение сервисов по проблемам эксплуатации, таких как скорость выпуска, имеет смысл в том случае, если этот критерий имеет большое значение. Аналогичным образом, часть системы может иметь экстремальные рабочие характеристики (такие, как масштабируемость). Разделение сервисов по оперативным целям позволит разработчикам отслеживать (с помощью функций пригодности) работоспособность и другие рабочие показатели сервиса.

Крупное разбиение сервисов означает устранение многих проблем координации, свойственных микросервисам, благодаря тому, что больше бизнес-контента остается внутри сервиса. Однако чем больше сервис, тем большее число эксплуатационных затруднений обостряется (еще один компромисс архитектуры).

Эволюция модульных взаимодействий

Миграция совместно используемых модулей (включая компоненты) является еще одной распространенной проблемой, с которой сталкиваются разработчики. Рассмотрим приведенную на рис. 6.3 структуру.

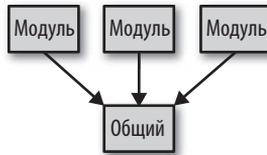


Рис. 6.3. Модули с эфферентной и афферентной связанностью

На рис. 6.3 все три модуля пользуются одной и той же библиотекой. Однако разработчикам необходимо разделить эти модули на отдельные сервисы. Как сервис сможет сохранить эту зависимость?

Иногда библиотеки могут быть разделены определенным образом, с сохранением отдельной функциональности, которая необходима каждому модулю. Рассмотрим приведенную на рис. 6.4 ситуацию.

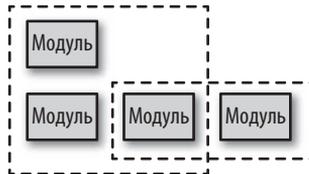


Рис. 6.4. Модули с общей зависимостью

На рис. 6.4 оба модуля нуждаются в конфликтующем модуле, который находится посередине. Если разработчикам повезет, то функциональность может определенным образом разбить на части среднюю би-

библиотеку, разделив совместно используемую библиотеку на те части, которые нужны каждому зависящему от нее модулю, как показано на рис. 6.5.

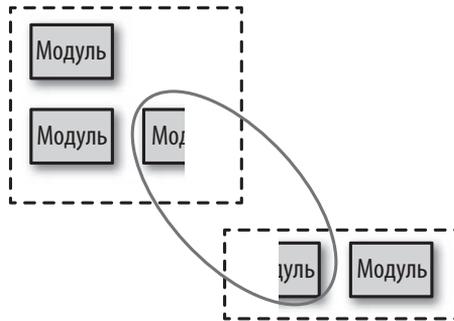


Рис. 6.5. Разделение общей зависимости

Однако вероятнее всего, что совместно используемую библиотеку нельзя будет так легко разделить на части. В этом случае разработчики могут извлечь этот модуль в совместно используемую библиотеку (например, JAR, DLL, gem или каким-либо другим механизмом) и использовать его из обоих мест, как показано на рис. 6.6.

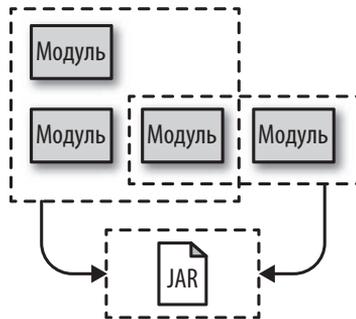


Рис. 6.6. Общая зависимость через файл JAR

Совместное использование является формой связанности, которую не рекомендуется применять в таких архитектурах, как микросервисы. Лучшим вариантом для совместно используемой библиотеки является дублирование, как показано на рис. 6.7.

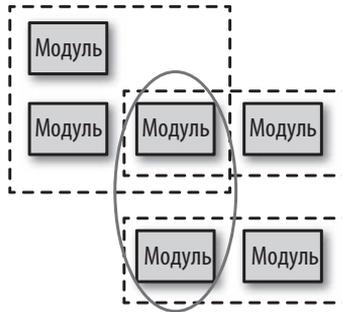


Рис. 6.7. Дублирование общей библиотеки для исключения точки связанности

В случае распределенной среды разработчики могут достичь такого же типа совместного использования с помощью обмена сообщениями или инициированием сервиса.

После того как разработчики идентифицировали правильный вариант разделения сервиса, на следующем шаге необходимо будет *отделить* предметную область от пользовательского интерфейса (UI). Даже в архитектурах микросервисов пользовательские интерфейсы часто решают перевести обратно в монолитную архитектуру; несмотря на это разработчики должны в определенной точке показать единообразный UI. Таким образом, разработчики обычно отделяют UI на раннем этапе миграции, создавая отображение прокси-слоя между компонентами пользовательского интерфейса и сервисами, предоставляемыми серверной частью системы, которую они вызывают. Отделение UI часто создает слой защиты от повреждений, изолируя изменения UI от изменений архитектуры.

Следующим шагом является *обнаружение сервисов*, которое позволяет сервисам находить и вызывать друг друга. В конце концов, архитектура будет состоять из сервисов, которые необходимо координировать. Построив механизм обнаружения на ранней стадии, разработчики могут медленно осуществлять миграцию тех частей системы, которые готовы к изменению. Разработчики часто применяют обнаружение сервиса как простой уровень прокси: каждый компонент вызывает посредника прокси, который, в свою очередь, выполняет отображение определенного применения.

Все проблемы в Computer Science можно решить другими уровнями косвенности, кроме, разумеется, проблемы слишком большого числа уровней косвенности.

— Дейв Уилер (*Dave Wheeler*)
и Кевлин Хенни (*Kevin Henney*)

Конечно, чем больше уровней косвенных средств добавят разработчики, тем более запутанным будет сервис навигации.

При миграции приложения из монолитной архитектуры в сервис-ориентированные архитектуры разработчик должен уделять пристальное внимание тому, как соединены модули в существующем приложении. Примитивное разделение вызывает серьезные проблемы с работоспособностью. Точки соединения в приложении становятся интегральными соединениями архитектуры с сопутствующими скрытыми состояниями, доступностью и прочими проблемами. Вместо того чтобы взяться за всю миграцию сразу, существует более прагматичный метод постепенной декомпозиции монолитной архитектуры на сервисы, учитывающий такие факторы, как границы транзакций, структурная связанность и прочие характеристики, чтобы создать несколько итераций реструктуризации. Прежде всего, следует разбить монолитную архитектуру на несколько крупных фрагментов приложений, зафиксировать точки интеграции, а затем повторить этот процесс. Для архитектуры микросервисов предпочтительна постепенная миграция.

При миграции монолитной архитектуры сначала постройте небольшое число крупных сервисов.

— Сэм Ньюмен (*Sam Newman*),
Построение микросервисов

Затем разработчики выбирают и отделяют из монолитной архитектуры выбранный сервис, закрепляя и вызывая точки. Функции при-

годности играют здесь важную роль. Разработчики должны построить функции пригодности, чтобы убедиться, что новые точки интеграции не меняются, и добавить контракты для пользователей.

Инструкции для построения эволюционирующей архитектуры

В этой книге мы уже использовали несколько биологических метафор, и вот еще одна. Наш мозг эволюционировал не в благоприятной, первозданной среде, в которой была тщательно построена каждая способность. Каждый слой базируется на лежащих ниже первозданных слоях. Основная часть нашего автоматического поведения (такого, как дыхание, возникающее чувство голода и т. д.) частично принадлежит нашему мозгу, который мало чем отличается от мозга рептилий. Вместо полноценной замены основных механизмов процесс эволюции создал новый слой сверху.

Архитектура программного обеспечения в крупных компаниях следует аналогичной схеме. Вместо рефакторинга каждой возможности большинство компаний использует все, что уже у них есть. Насколько мы любим рассуждать об архитектуре с первозданными, идеализированными настройками, в такой же степени реальный мир вносит путаницу из технических недоработок, конфликтов интересов и ограниченного бюджета. Архитектура в крупных компаниях похожа на мозг человека — системы низкого уровня все еще занимаются важными деталями связующего программного обеспечения, но при этом имеют некоторый набор старых убеждений. Компании не любят выводить из эксплуатации то, что еще работает, и тем самым способствуют появлению проблем с интеграцией архитектуры.

Настройка возможности эволюционирования в существующей архитектуре проблематична, так как если разработчики никогда не вносили в архитектуру изменения, маловероятно, что они появятся спонтанно. Ни один разработчик, каким бы талантливым он ни был, не сможет без огромных усилий преобразовать большой комок грязи в современную архитектуру микросервисов. К счастью, проекты могут использовать

преимущества без изменения их внутренней архитектуры, построив несколько точек гибкости в существующую архитектуру.

Удаление ненужной изменчивости

Одной из целей непрерывной поставки является стабильность, построенная на хорошо зарекомендовавших себя частях. Простым проявлением этой цели является точка зрения на практики DevOps как на инструмент строительства неизменной инфраструктуры. Мы обсуждали динамическое равновесие экосистемы разработки программного обеспечения в главе 1, где как нигде очевидно, насколько велики основные сдвиги зависимостей программного обеспечения. Системы программного обеспечения претерпевают постоянное изменение по мере обновления разработчиками возможностей, выпуска пакетов обновлений и обычной настройки их программного обеспечения. Операционные системы являются хорошим примером, потому что они претерпевают постоянные изменения.

С помощью современных практик DevOps удается локально решить проблему динамического равновесия, заменив схему *снежинки* (snowflakes)¹ на *неизменяемую инфраструктуру*. Компьютеры со схемой *снежинки* выкраивались вручную персоналом по операциям, и все последующее обслуживание также выполнялось вручную. Чед Фаулер придумал термин *неизменная инфраструктура* (immutable infrastructure) в посте *Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components* в своем блоге². Неизменная инфраструктура относится к системам, полностью определяемым программно. Все изменения в этой системе должны происходить с помощью исходного кода, а не путем изменений работающей операционной системы. Таким образом, вся система оказывается неизменной с точки зрения операционной системы — после использования текущего варианта системы никаких других изменений не происходит.

¹ https://ru.wikipedia.org/wiki/Схема_снежинки

² <http://chadfowler.com/2013/06/23/immutable-deployments.html>

В то время как неизменность может звучать как противоположность эволюционированию, справедливо как раз противоположное. Системы программного обеспечения состоят из тысячи движущихся частей, и все они взаимно заблокированы тесными зависимостями. К сожалению, разработчики все еще борются с непредвиденными побочными эффектами изменений одной из частей. С помощью ограничения возможности неожиданных изменений мы контролируем большое число факторов, которые делают систему хрупкой. Разработчики стремятся заменить переменные в коде постоянными, чтобы сократить векторы изменений. DevOps ввела эту концепцию для операций, делая их более декларативными.

Неизменная инфраструктура следует нашей рекомендации *убрать ненужные переменные*. Построение программного обеспечения, которое эволюционирует, означает управление как можно большим числом неизвестных факторов. Практически невозможно построить функции пригодности, которые могут предвидеть, как последний пакет обновлений может влиять на приложение. Вместо этого разработчики строят инфраструктуру заново всякий раз при выполнении конвейера развертывания, отлавливая как можно активнее разрушающие изменения. Если разработчики могут как можно скорее убрать известные основополагающие и способные к изменению части, такие как операционная система, у них будет меньше нагрузки на тестирование.

Архитекторы могут найти все виды способов преобразования переменных в константы. Многие команды также распространили рекомендации по неизменной инфраструктуре на разработку среды. Сколько раз члены команд восклицали: «Но это работает на моей машине!» За счет того, что каждый разработчик имеет в точности одно и то же изображение, пропадает масса ненужных переменных. Например, большинство команд разработчиков автоматизируют обновление библиотек разработок с помощью репозитория. А что насчет обновления такого инструмента, как IDE? Фиксируя среду разработок в качестве неизменной инфраструктуры, разработчики всегда работают на одной и той же основе.

Построение неизменной среды разработок также позволяет распространять полезные инструменты в выполняемых проектах. Парное программирование, включая парную ротацию, является распространенной практикой во многих гибких, ориентированных на проектирование командах разработчиков, при этом каждый член команды регулярно подменяется с периодичностью от нескольких часов до нескольких дней. Но когда инструмент, которым разработчик пользовался на своем компьютере вчера, отсутствует сегодня — это вызывает огорчение. Подготовив один источник для всех систем разработчика, становится легче добавлять полезные инструменты во все системы сразу.

ОПАСНОСТИ СНЕЖИНОК

Пост под названием «Knightmare: A DevOps Cautionary Tale»¹ в популярном блоге служит поучительной историей про серверы снежинок. Компания, предоставляющая финансовые услуги, раньше использовала алгоритм PowerPeg, который обрабатывал детали торговых операций, но уже несколько лет этот код не использовался. Но разработчики никогда не удаляли код. Он оставался под управлением переключателя функций, который оставался отключенным. Благодаря периодическим изменениям разработчики реализовали новый торговый алгоритм, который получил название SMARS. Но поскольку они были ленивыми, то решили повторно воспользоваться флажком функции PowerPeg, чтобы реализовать новый код SMARS. 1 августа 2012 года разработчики развернули новый код на семи серверах. К сожалению, их система запустилась на восьми серверах, и один из них не был обновлен. Когда они включили переключатель функций PowerPeg, семь серверов начали совершать продажи... а остальные — покупки! Разработчики случайно задали наихудший сценарий рынка — они автоматически начали продавать по низкой цене, а покупать по высокой. Уверенные в том, что виной всему был новый код, разработчики откатили его назад на семи серверах, но оставили включенным переключатель функций, что привело к тому, что код PowerPeg выполнялся на всех семи серверах. Им потребо-

¹ <http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale>

валось 45 минут, чтобы обуздать хаос, принесший потери свыше 400 млн долларов США. К счастью, их спас бизнес-ангел (частный венчурный инвестор, обеспечивающий финансовую и экспертную поддержку компаний), так как вся эта сумма была больше, чем общая стоимость их компании.

Эта история выделяет проблемы с неизвестной варьируемостью. Повторное использование старого флага функции было необдуманным поступком, так как наилучшим вариантом было бы удалить их сразу после того, как их назначение было выполнено. Не использовать автоматизацию важного программного обеспечения для серверов также считается необдуманным поступком в современной среде DevOps.



Выявите и удалите ненужные переменные.

Сделайте решения обратимыми

Неизбежно, что активно эволюционирующие системы потерпят неудачу непредвиденным образом. Когда эти сбои начнутся, разработчикам необходимо будет создать новые функции пригодности, чтобы предотвратить сбои в будущем. Но как вы восстановитесь после сбоя?

Существуют многочисленные практики DevOps, которые позволяют использовать *обратимые решения* (reversible decisions), то есть такие, которые необходимо отменить. Например, *синие/зеленые развертывания*, при которых операции имеют две идентичные экосистемы (вероятно, виртуальные), а именно *синие* и *зеленые*, что распространено в DevOps. Если текущая промышленная система работает как *синяя*, тогда *зеленая* подготавливает область для следующего релиза. Когда *зеленая* версия готова, она становится промышленной системой, *синяя* временно переходит в состояние резервной копии. Иногда с *зеленой* системой все идет не так, как надо, тогда операции могут вернуться назад в *синюю* систему без особых неудобств. Если с *зеленой* системой

все в порядке, *синяя* становится подготовительной площадкой для следующего релиза.

Переключатели функций являются еще одним распространенным способом для разработчика сделать решения обратимыми. Благодаря разворачиванию изменений под контролем переключателей функций разработчики могут осуществлять их для небольшой группы пользователей (операция получила название *canary releasing*, martinfowler.com/bliki/CanaryRelease.html), чтобы оценить изменение. Если функция ведет себя непредсказуемо, разработчики могут переключиться назад в исходное состояние и устранить сбой, прежде чем попытаются обновить систему снова. Следует убедиться, что устаревшие версии удалены!

Применение переключателя функций значительно уменьшает риск в этих сценариях. Прокладка маршрута к сервису или прокладка маршрута к определенному экземпляру сервиса на основании контекста запроса является еще одним распространенным методом для раннего релиза в экосистемах микросервисов.



Сделать как можно больше решений обратимыми (без чрезмерного проектирования).

Предпочтение следует отдавать эволюционированию, а не предсказуемости

...есть известное знакомое, то есть вещи, о которых мы знаем, что мы о них знаем. Мы знаем также, что есть известное неизвестное, то есть, говоря иначе, мы знаем, что есть вещи, о которых мы не знаем. Но есть также неизвестное неизвестное — те вещи, о которых мы не знаем, что мы о них не знаем.

— *Бывший министр обороны США Дональд Рамсфельд (Donald Rumsfeld)*

Неизвестные неизвестные являются заклятыми врагами систем программного обеспечения. Многие проекты начинаются с составления списка *известных неизвестных*: то, о чем разработчики знают, что они должны узнать в отношении областей и технологий. Однако проекты также становятся жертвами *неизвестных неизвестных*: то, о чем никто не знает и что появляется неожиданно и внезапно. Вот почему страдают все усилия при попытке масштабного проектирования (Big Design Up Front) — разработчики не могут проектировать для неизвестных неизвестных.

Все архитектуры из-за *неизвестных неизвестных* становятся итеративными; гибкие компании просто распознали и сделали это раньше.

— *Марк Ричардс (Marc Richards)*

Несмотря на то что ни одна архитектура не может пережить неизвестное, мы знаем, что динамическое равновесие становится предсказуемо бесполезным в программном обеспечении. Вместо этого мы предпочитаем встраивать в программное обеспечение *способность эволюционировать*: если проекты могут легко принимать изменения, разработчикам не нужен магический кристалл. Архитектура не является исключительно предварительной активностью — проекты на протяжении своего существования непрерывно меняются как явным, так и неожиданным образом. Одним из способов, часто используемым разработчиками для изоляции от изменений, является *уровень защиты от повреждений* (anticorruption layer).

Построение уровня защиты от повреждений

Проекты часто нуждаются в связи с библиотеками, которые предоставляют связующее программное обеспечение: очередь сообщений, поисковые системы и т. п. *Антипаттерн Абстракция отвлечения* описывает сценарий, в котором проект слишком сильно «привязывается» к внешней библиотеке, либо к коммерческой, либо свободного доступа.

После того как подходит время для обновления или переключения библиотеки, большая часть приложений, использующих библиотеку, внедряет предположения, основанные на предыдущих абстракциях библиотеки. Предметно-ориентированное проектирование включает защиту от этого явления под названием *уровень защиты от повреждения*. Вот пример.

Гибкие компании разработчиков оценили принцип *последнего ответственного момента* при принятии решений, который используется для противодействия распространенной угрозе в проектах, связанной со слишком ранним приобретением сложных систем. Мы периодически выполняли работы по проекту на Ruby on Rails для заказчика, который управлял оптовыми продажами автомобилей. После того как приложение стало функционировать, возник неожиданный рабочий поток. Оказалось, что дилеры подержанных автомобилей стремились выложить на сайт аукциона крупные партии новых автомобилей (как по количеству автомобилей, так и по числу фотографий для одного автомобиля). Мы поняли, что насколько люди не доверяют продавцам подержанных автомобилей, настолько же продавцы *действительно* не доверяют друг другу; поэтому каждый автомобиль должен иметь фотографии с изображением каждой молекулы этого автомобиля. Пользователям нужен способ загрузки на сайт с визуализацией прогресса через какой-либо механизм пользовательского интерфейса, например, индикатора выполнения, либо они могли бы проверить загрузку позже, чтобы узнать, загрузилась ли партия. В переводе на технические термины им была нужна асинхронная загрузка.

Очередь сообщений является одним из традиционных архитектурных решений этой проблемы, и команда обсуждала, добавлять ли в архитектуру открытый доступ к очереди. Распространенной ловушкой в связи с этим для многих проектов является положение «Мы знаем, что в конце концов нам потребуется очередь сообщений для многочисленного персонала, поэтому давайте возьмем одного с богатым воображением и позже обучим его этой работе». Проблемой такого подхода является *технический долг*: то, что является частью выполняемого проекта, и то, что не предполагалось использовать здесь, стоит

на пути того, что должно здесь оказаться. Большинство разработчиков рассматривают старый неработоспособный код как единственную форму технического долга, однако проекты также могут по неосторожности *приобрести* технический долг под видом преждевременной сложности (premature complexity).

Для таких проектов разработчик архитектуры поощряет разработчиков программного обеспечения найти более простой путь. Один разработчик обнаружил чрезвычайно простой инструмент BackgroundDRb (<https://github.com/gnufied/backgroundrb>) в библиотеке с открытым исходным кодом, с помощью которого можно имитировать единичную очередь сообщений с резервной копией в реляционной базе данных. Разработчик знал, что этот простой инструмент, вероятно, никогда не будет масштабироваться в другие проблемы, и других возражений не было. Вместо того чтобы пытаться предусмотреть использование в будущем, разработчик обеспечил возможность замены этого инструмента, поместив его позади API. В *последнем ответственном моменте* содержатся ответы на такие вопросы: «Должен ли я принимать это решение сейчас?», «Есть ли способ безопасно отложить это решение, не замедляя выполнение какой-либо работы?» и «Что я могу создать сейчас, и этого будет достаточно, но при этом смогу при необходимости все это легко заменить?»

Примерно год спустя появился второй запрос для асинхронности в форме контролируемых по времени событий в отношении продаж. Разработчик архитектуры оценил ситуацию и решил, что второго экземпляра BackgroundDRb будет достаточно, установил его на место и забыл. Примерно два года спустя появился третий запрос для постоянно обновляемых величин, таких как кэши и сводки. Команда поняла, что текущее решение не сможет управлять новой рабочей нагрузкой. Однако теперь у них появилась хорошая идея о том, какой тип асинхронного поведения необходим для этого приложения. С этого момента проект переключился на Starling (<https://github.com/starling/starling>), простую, но более традиционную очередь сообщений. Поскольку используемое решение было изолировано интерфейсом, потребовалась всего пара разработчиков и менее одной итерации

(одна неделя в этом проекте), чтобы завершить переход, не отвлекая от работы других разработчиков проекта.

Поскольку разработчик архитектуры поместил уровень защиты от повреждений вместе с интерфейсом, замена одной части функциональности становится настоящим упражнением в механике. Построение уровня защиты от повреждений вдохновляет разработчика архитектуры подумать о *семантике* того, что им необходимо из библиотеки, а не о *синтаксисе* конкретного API. Но это не повод *абстрагировать все!* Некоторым сообществам разработчиков нравятся приоритетные слои абстракций для определенной степени отвлеченности, но при понимании потерь, когда вы должны будете обратиться в **Factory**, чтобы получить *проху* доступа к удаленному интерфейсу **Thing**. К счастью, большинство современных языков и IDE позволяют разработчикам оказаться *как раз вовремя* при извлечении интерфейсов. Если проект оказывается привязанным к устаревшей библиотеке, которая нуждается в изменении, IDE может *извлечь интерфейс* от лица разработчика, создав точно в срок (Just in time) уровень защиты от повреждений.



Создавайте точно в срок уровни защиты от повреждений для изоляции от изменений библиотеки.

Контроль точки связанности в приложениях, особенно для внешних ресурсов, является одной из ключевых обязанностей разработчика архитектуры. Старайтесь найти необходимое время для того, чтобы добавить зависимости. Как архитектор, помните, что зависимости обеспечивают преимущества, но также налагают ограничения. Убедитесь, что эти преимущества перевешивают затраты на обновления, управление зависимостями и т. п.

Разработчики понимают преимущества всего и не принимают никаких компромиссов!

— Рич Хики (*Rich Hickey*), создатель языка *Clojure*

Разработчики архитектуры должны понимать как преимущества, так и компромиссы и соответствующим образом строить процесс проектирования.

Использование уровней защиты от повреждений способствует эволюционированию. Несмотря на то что архитекторы не могут прогнозировать будущее, мы можем, по крайней мере, снизить затраты на изменения так, чтобы они не оказывали негативного влияния.

Практический пример: шаблоны сервисов

Архитектуры микросервисов предназначены для архитектур *без разделения ресурсов* — каждый компонент несвязан насколько это возможно с остальными компонентами, в соответствии с принципом ограниченного контекста. Однако исключение связанности между сервисами относится прежде всего к классам областей, схемам баз данных и к остальным точкам связанности, которые вредят способности эволюционировать. Команды разработчиков часто стремятся управлять некоторыми аспектами однородности технической связанности — придерживаясь нашей рекомендации *удалить ненужные переменные* для обеспечения однородности. Например, мониторинг, сбор данных и прочие средства диагностики очень важны в архитектуре этого типа благодаря изобилию движущихся частей. Когда операции должны управлять тысячами сервисов, а команды обслуживания забывают добавить и контролировать возможности своих сервисов, результат может быть катастрофическим. Сразу после развертывания этот сервис исчезнет в черной дыре, потому что в этих средах, если их нельзя контролировать, он невидимый. И все же как команды могут принудительно обеспечить согласованность в средах с низкой степенью связанности?

Шаблоны сервисов являются распространенными решениями для обеспечения согласованности. Используются предварительно сконфигурированные наборы общих библиотек инфраструктур, таких как исследование сервиса, мониторинг, сбор данных, метрики, аутентификация/авторизация и т. п. В крупных организациях команда по

совместному использованию инфраструктуры управляет шаблонами сервисов. Команды реализации сервисов используют шаблон в качестве автоматической кодогенерации, описывая внутри него поведение сервисов. Если для сбора данных необходимо обновление, команда совместного использования инфраструктуры может управлять ею независимо от команд обслуживания — они никогда не будут знать (или следить за этим), что изменение произошло. Если произошло разрушительное изменение, оно перестает действовать во время фазы инициализации конвейера развертывания, предупреждая как можно раньше разработчиков о возникшей проблеме.

Это хороший пример того, что мы имеем в виду, когда говорим о *надлежащей связанности*. Дублирование функциональности архитектуры во всех сервисах создает множество хорошо известных проблем. Найдя необходимый уровень связанности, мы можем свободно эволюционировать, не создавая новых проблем.



Воспользуйтесь шаблонами сервисов, чтобы связать вместе надлежащие части архитектуры — элементы инфраструктуры, которые дают возможность командам воспользоваться преимуществами связанности.

Шаблоны сервисов чрезвычайно адаптируемы. Исключение технической архитектуры как основной структуры системы облегчает настройку изменений только на этот размер архитектуры. Когда разработчики создают слоистую архитектуру, изменение легко проводить в пределах каждого слоя, и оно приводит к высокой степени связанности. Несмотря на то что слоистая архитектура разделяет все части технической архитектуры, она запутывает другие проблемы, такие как проблемы областей, безопасности, операций и т. п. Построив часть архитектуры только для одной технической архитектуры (например, шаблоны сервисов), разработчики могут изолировать и объединить изменение для всей области целиком. В главе 4 мы рассматривали, как элементы архитектуры следует представлять в качестве развертываемых блоков.

Построение жертвенной архитектуры

В своей книге *Мифический человек-месяц* Фред Брукс (Mythical Man Month, Fred Brooks) говорит о том, когда строится новая система программного обеспечения.

Вопрос управления, таким образом, заключается не в том, надо ли создавать пилотную систему и выбрасывать ее. Вы и так это сделаете... Поэтому планируйте выбросить ее с самого начала; все равно так оно и получится.

— *Фред Брукс (Fred Brooks)*

Точка зрения автора состояла в том, что после того, как команда построила систему, они знают все *неизвестные неизвестные* и надлежащие решения по архитектуре, которые сначала никогда не понятны. Только следующие версии получают преимущества от этих уроков. На уровне архитектуры разработчики ведут борьбу, чтобы планировать радикально меняющиеся требования и характеристики. Одним из способов, позволяющих узнать все необходимое, чтобы выбрать правильную архитектуру, является разработка доказательства правильности концепции. Мартин Фаулер определил жертвенную архитектуру (sacrificial architecture)¹, как архитектуру, предназначенную, чтобы ее выбросить, если концепция доказала свою успешность. Например, компания eBay начала свою работу в 1995 году с набора скриптов на Perl, размещенных на сайте; в 1997 году они были переведены на C++, а затем в 2002 году — на Java. Очевидно, что компания eBay имела ошеломляющий успех несмотря на то, что им пришлось несколько раз менять архитектуру. Социальная сеть «Твиттер» — еще один удачный пример успешного применения этого метода. Когда «Твиттер» появился, он был написан на Ruby и использовал фреймворк Ruby on Rails, чтобы ускорить время выхода на рынок. Однако по мере того как «Твиттер» становился популярным, платформа уже не могла поддерживать этот масштаб, что

¹ <https://martinfowler.com/bliki/SacrificialArchitecture.html>

приводило к частым отказам и к ограничению доступа. Многие пользователи этой сети слишком хорошо были знакомы с их сообщением об ошибке, представленным на рис. 6.8.

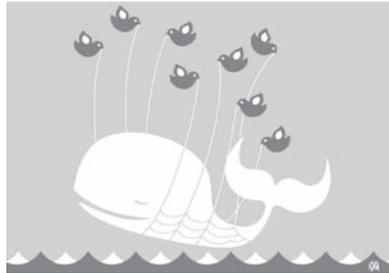


Рис. 6.8. Знаменитое сообщение об ошибке «Твиттера»

Поэтому «Твиттер» реконструировал свою архитектуру, чтобы заменить бэкенд на нечто более надежное. Но можно было поспорить, что именно эта тактика явилась причиной выживания компании. Если бы инженеры «Твиттера» построили надежную платформу с самого начала, это привело бы к значительному времени задержки их выхода на рынок, достаточной для *Snitter* или другой альтернативной сети, которая вышла бы на рынок раньше. Несмотря на все сложности начало работы с жертвенной архитектурой в конце концов окупается.

Облачные среды сделали жертвенную архитектуру еще более привлекательной. Если у разработчиков есть проект, который необходимо испытать, то построение первоначальной версии в облаке значительно сокращает ресурсы, требуемые для релиза программного обеспечения. Если проект успешный, разработчики архитектуры могут позволить себе потратить определенное время, чтобы построить более подходящую архитектуру. Если разработчики заботятся об уровнях защиты от повреждений и других практиках эволюционирующей архитектуры, они могут уменьшить сложности миграции.

Многие компании создают жертвенные архитектуры для достижения минимально жизнеспособного продукта¹, чтобы убедить рынок в его существовании. Несмотря на то что это хорошая стратегия, команда

¹ https://ru.wikipedia.org/wiki/Минимально_жизнеспособный_продукт

в конечном счете распределяет время и ресурсы, чтобы построить более устойчивую архитектуру, в надежде на то, что она будет менее заметной, чем у «Твиттера».

Еще одним аспектом влияния технического долга на многие первоначально успешные проекты (которые снова были проанализированы Фредом Бруксом, когда он обращается к *эффекту второй системы*) является тенденция небольших, элегантных и успешных систем эволюционировать в гигантских, нагруженных функциями монстров, и все из-за завышенных ожиданий. Представители бизнеса не любят отказываться от функционирующих программ, поэтому архитектура стремится к тому, чтобы что-то можно было добавить, ничего не удаляя и не выводя из эксплуатации.

Технический долг эффективно работает как метафора, так как он резонирует с опытом проектирования и представляет недостатки проектирования, независимо от лежащих в основе движущих сил. Технический долг обостряет проблемы с ненадлежащей связанностью в проектах — плохое проектирование часто проявляется как патологическая связанность — и с другими антипаттернами, которые затрудняют реконструкцию программы. По мере того как разработчики реконструируют архитектуру, их первым шагом должно быть удаление исторически сложившихся компромиссов проектирования, проявляющихся как технический долг.

Уменьшить внешние изменения

Распространенной особенностью каждой платформы разработок являются *внешние зависимости*: инструменты, объекты структуры, библиотеки и прочие средства, предоставляемые и обновляемые (что важнее) через интернет. Разработка программного обеспечения основывается на стеке башни абстракций, каждая из которых построена на предыдущих абстракциях. Например, операционные системы относятся к категории внешних зависимостей, находящихся за пределами контроля разработчика. Пока компании не захотят написать свою собственную операционную систему и все остальные вспомогательные программы, они должны полагаться на внешние зависимости.

Большинство проектов полагается на огромный массив компонентов третьих сторон, применяемых с помощью разработки инструментов. Разработчикам нравятся зависимости, потому что они дают преимущества, однако многие разработчики игнорируют тот факт, что все зависимости несут затраты. Если полагаться на код сторонней организации, то разработчикам необходимо создать собственные средства безопасности: разрушающих изменений, удалений без предупреждения и т. п. Управление этими внешними частями проектов очень важно для создания эволюционирующей архитектуры.

ОДИННАДЦАТЬ СТРОК КОДА, КОТОРЫЕ СЛОМАЛИ ИНТЕРНЕТ

В начале 2016 года разработчики JavaScript усвоили суровый урок об угрозах, зависящих от тривиальных вещей. Разработчик, создавший большое количество небольших утилит, стал проявлять недовольство тем, что один из его модулей столкнулся с коммерческим проектом разработки программного обеспечения, разработчики которого попросили переименовать этот модуль. Вместо того чтобы выполнить эту просьбу, он удалил более 250 своих модулей, включая одну библиотеку `left pad.io` — одиннадцать строк кода для заполнения строк нулями или пробелами (если 11 строк кода можно назвать библиотекой). К сожалению, многие крупные проекты на JavaScript (включая `node.js`) основывались на этой зависимости. Когда этот код пропал, все развернутые на JavaScript проекты были выведены из строя.

Администратор репозитория, занимающийся пакетами для JavaScript, предпринял неожиданное действие по восстановлению этой программы, чтобы восстановить экосистему, но это породило в сообществе более глубокое обсуждение относительной разумности тенденций в отношении управления зависимостями.

Из этой истории можно извлечь два ценных для разработчиков архитектуры урока. Первый урок: следует помнить о внешних библиотеках, предоставляющих *одновременно* преимущества и затраты. Убедитесь в том, что преимущества оправдывают затраты. Второй урок: не позволяйте внешним силам влиять на стабильность ваших сборок. Если ранее требуемая зависимость внезапно пропала, необходимо отклонить это изменение.

Эдсгер Дейкстра, легендарная личность в области Computer Science, в 1968 году написал «О вреде оператора Go To», где раскритиковал существующую практику бесструктурного написания кода, что в итоге привело к революции структурного программирования. Начиная с этого времени, «считавшиеся вредными» стали метафорой в разработке программного обеспечения.

Управление переходной зависимостью — наш «считавшийся вредным» момент.

— Крис Форд (*Chris Ford*) (*не родственник Нула*)

Позиция Криса состоит в том, что, пока мы осознаем серьезность проблемы, мы не можем определить решение. Пока мы не предлагаем решения проблемы, нам необходимо выделить ее, потому что она значительно влияет на эволюционирование архитектуры. Стабильность является одной из основ как непрерывной поставки, так и эволюционирующей архитектуры. Разработчики не могут разработать повторяемой практики проектирования, которая могла бы справиться с неопределенностью. Разрешение третьим сторонам вносить изменения в основные зависимости бросает вызов этому принципу.

Мы рекомендуем разработчикам использовать превентивный метод управления зависимостями. Хорошим началом в управлении зависимостями является моделирование зависимостей с использованием модели *вытягивания* (*pull model*). Например, зададим внутренний репозиторий с контролем версий, который действовал бы как хранилище компонентов сторонней организации и рассматривал изменения со стороны внешнего мира как запрос на включение внесенных изменений в этот репозиторий. Если появились благоприятные изменения, их можно допустить в экосистему. Однако если ключевые зависимости внезапно пропали, этот запрос на внесение изменений следует отклонить как дестабилизирующую силу.

С помощью принципа непрерывной поставки репозиторий компонентов сторонней организации использует свой собственный конвейер

развертывания. При возникновении обновления конвейер развертывания включает в состав архитектуры это изменение, затем выполняет сборку и тестирует состояния затронутых приложений. При успешном тестировании это изменение допускают в экосистему. Таким образом, сторонние зависимости применяют ту же практику и механизмы проектирования внутреннего развития, с пользой размывая линии часто не имеющих значения различий между написанным в компании кодом и зависимостями, внесенными сторонними организациями — в конце концов, это все код, используемый в проекте.

Обновление библиотек и фреймворков

Разработчики архитектуры делают общее различие между *библиотеками* и *фреймворками*, говоря, что «разработчик вызывает библиотеку, тогда как фреймворк вызывает разработчика». В общем случае, разработчики составляют подкласс фреймворков (которые, в свою очередь, вызывают эти производные классы), поэтому различие в том, что фреймворк вызывает код. И наоборот, код библиотеки обычно приходит в виде коллекции связанных классов и/или функций, которые разработчик вызывает при необходимости. Поскольку фреймворк вызывает код разработчика, это создает высокую степень связанности с фреймворком. Сравните это с кодом библиотеки, которая содержит обычно более утилитарный код (такой, как XML-парсеры, сетевые библиотеки и т. п.) и имеет низкую степень связанности.

Мы предпочитаем библиотеки, потому что они вносят меньше связанности в используемые приложения, что облегчает их выгрузку, когда технической архитектуре необходимо эволюционировать.

Одна причина рассматривать библиотеки и фреймворки по-разному перешла в практику проектирования. Фреймворки включают такие возможности, как UI, средство объектно-реляционного отображения, автоматическую кодогенерацию, такую как модель–представление–поведение, и т. п. Поскольку фреймворк формирует кодогенерацию для остальных приложений, все программы в приложении подвергаются воздействию, вызванному изменениями фреймворка. Многие из нас остро почувствовали эти затруднения, так как в любое время

команде теперь можно выполнять обновление фундаментального фреймворка более чем двумя главными версиями, и усилия, затрачиваемые для окончательного обновления, стали неимоверными.

Поскольку фреймворки являются основополагающими частями приложений, команды должны проявлять активность в отношении следующих обновлений. Библиотеки обычно образуют менее хрупкие точки связанности, чем фреймворки, предоставляя командам возможность быть несколько небрежными в отношении обновлений. Одна неофициальная модель управления рассматривает обновление фреймворков как *выталкиваемые* обновления, а обновления библиотек как *вытягиваемые* обновления. Когда обновляется основополагающий фреймворк (один из тех, чьи афферентные/эфферентные числа связанностей выше определенного порога), команды должны применить обновление сразу после того, как новая версия станет стабильной и команда сможет распределить время для изменения. Несмотря на то что это потребует времени и усилий, потраченное время относится к части затрат, если команда бесконечно откладывает обновление.

Поскольку большинство библиотек обеспечивает утилитарные функции, команды могут позволить себе обновлять их только тогда, когда появляется новая требуемая функция, используя модель «обновляй когда нужно».



Обновлять зависимости фреймворка следует активно; обновлять библиотеки следует пассивно.

Отдавайте предпочтение непрерывной поставке, а не снимкам состояния системы

Многие инструменты управления зависимостью применяют технику под названием *снимки состояния системы*, или *снэпшоты* (snapshots) для моделирования развития на ходу. Создание снэпшотов изначально предназначалось для индикации компонентов практически готовых для релизов, но все еще находящихся в разработке, при этом подразумевалось, что код может изменяться на регулярной основе. После того

как компонент был «благословлен» с присвоением номера версии, название «СНЭПШОТ» можно убрать.

Разработчики используют снимки, потому что исторически сложилось мнение, что тестирования трудно проводить и они занимают много времени. Это привело разработчиков к попытке отделить то, что меняется, от того, что не меняется.

Как мы ожидаем, в эволюционирующей архитектуре все время все должно меняться, и составляем практику проектирования и функции пригодности для размещения изменений. Например, когда проект имеет превосходный охват тестами и конвейер развертывания, разработчики проверяют каждое изменение для каждого компонента с помощью автоматического конвейера развертывания. У разработчиков нет причин содержать «специальный» репозиторий для каждой части проекта.



Для внешних зависимостей следует отдавать предпочтение непрерывной поставке, а не снимкам.

Снимки являются артефактом из эры разработок, когда всесторонние тестирования не были распространены, хранение данных было дорогостоящим, а проверки — трудновыполнимыми. Сегодня обновленная практика проектирования исключает неэффективное управление зависимостями компонентов.

Непрерывная поставка предлагает эффективный метод анализа зависимостей, который излагается в этой книге. В настоящее время разработчики имеют дело только со *статическими* зависимостями, связанными по номерам версий, собранными в виде медиаданных в файле системы сборки. Однако этого недостаточно для современных проектов, для которых необходим механизм указания *теоретического обновления* (speculative updating). Таким образом, как предлагается в этой книге, разработчики должны ввести два новых обозначения для внешних зависимостей: *подвижная* (fluid) и *охраняемая* (guarded). *Подвижные* зависимости стремятся обновить себя до следующей версии автоматически, используя для этого такие механизмы, как конвейер-

еры развертывания. Например, пусть этот порядок действий плавно опирается на версию фреймворка 1.2. Когда фреймворк обновит себя до версии 1.3, это изменение будет внесено в систему через конвейер развертывания, который настроен на пересборку проекта всякий раз при изменении любой его части. Если конвейер развертывания подходит к завершению, подвижная зависимость между компонентами обновлена. Однако если что-то препятствует успешному завершению, например неудачное тестирование, нарушенная зависимость алмазов¹ (diamond dependency) или другие проблемы, зависимость обновляется до *охраняемой* на фреймворке версии 1.2, это означает, что разработчик должен попытаться определить и устранить проблему, восстановив подвижную зависимость. Если компонент оказался действительно несовместимым, разработчик создает постоянную статическую ссылку на старую версию, избегая последующих автоматических обновлений.

Ни один из популярных инструментов сборки не поддерживает этот уровень функциональности, поэтому разработчики должны выполнить сборку этого монитора поверх существующих инструментов сборки. Однако эта модель зависимостей чрезвычайно хорошо работает в эволюционирующих архитектурах, в которых время цикла является важной основополагающей величиной, будучи пропорциональной многим другим ключевым показателям.

Версии внутренних сервисов

В любой архитектуре интеграции разработчики неизбежно должны использовать конечные точки системы управления версиями для оценки эволюционирования поведения системы. Разработчики используют два распространенных шаблона для конечных точек версии, *нумерации версии* и *внутреннего разрешения*. Для нумерации версий разработчики предложили имя новой конечной точки, часто включа-

¹ Возникает, когда зависимость компонентов представляет собой граф типа Алмаз. Если пронумеровать вершины от 1 до 4, то в этом графе последняя вершина 4 является компонентом ПО, который для правильной работы использует связь с компонентами во 2 и 3 вершинах. Нарушенная зависимость алмаза возникает тогда, когда мы обновили компонент 4 через компонент 3, но не обновили зависимость между 4 и 2 компонентами. — *Примеч. науч. ред.*

ющей номер версии, когда возникает заметное изменение. Это дает возможность старым точкам интеграции вызывать унаследованную версию, при этом более новые точки вызывают новые версии. Альтернативой является внутреннее разрешение, при котором вызывающие объекты никогда не меняют конечную точку, а вместо этого разработки встраивают в эту конечную точку логику для определения контекста вызываемого объекта, возвращая правильную версию. Преимуществом сохранения имени является меньшая связанность с номерами определенных версий в вызове приложений.

В других случаях это значительно ограничивает число поддерживаемых версий. Чем больше версий, тем больше проверок и прочих нагрузок на проектирование. Рекомендуется поддерживать только две версии за раз и только временно.



Когда сервис контролируют версии, предпочтительнее использовать внутренний контроль версий для нумерации; поддерживать за раз рекомендуется только две версии.

Практический пример: эволюционирование рейтингов PenultimateWidgets

Компания PenultimateWidgets использует архитектуру микросервисов, поэтому разработчики могут вносить небольшие изменения. Давайте поближе рассмотрим все детали этих изменений, переключив звездочную систему оценок, как было показано в главе 3. В настоящее время компания PenultimateWidgets использует сервис звездочного рейтинга, чьи составляющие показаны на рис. 6.9.

Как показано на рис. 6.9, сервис звездочного рейтинга состоит из базы данных и слоистой архитектуры, системы хранения данных, бизнес-правил и пользовательского интерфейса. Не все микросервисы PenultimateWidgets содержат пользовательский интерфейс. Некоторые сервисы, главным образом, информационные, а остальные, имеющие

пользовательский интерфейс, тесно связаны с характеристиками сервисов, как в случае с сервисом звездочного рейтинга. База данных традиционно является реляционной, включающей столбик для отслеживания рейтингов определенного элемента пользовательского интерфейса.

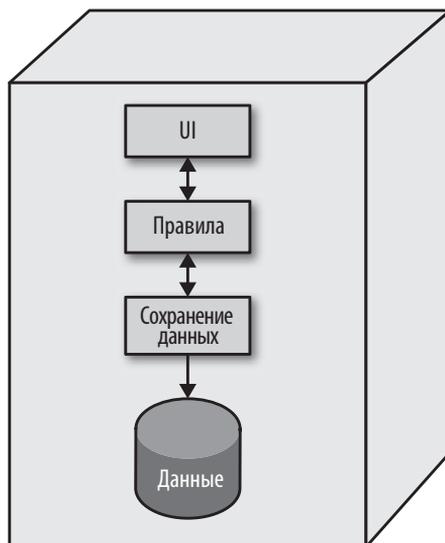


Рис. 6.9. Внутренние элементы звездочного сервиса Widgetco

Когда группа принимает решение обновить сервис, чтобы поддержать полужвездочные рейтинги, они изменяют исходный сервис, как это показано на рис. 6.10.

На рис. 6.10 в базу данных добавлен новый столбец для управления дополнительными базами данных, если рейтинг дополнительно имеет полужвездочные оценки. Разработчики архитектуры также добавили в наш сервис компонент-посредник для разрешения различий возврата на границе сервиса. Вместо того чтобы форсировать сервис вызова и разобраться с номерами версий этого сервиса, сервис звездочного рейтинга определяет тип запроса и отправляет назад данные, в каком бы формате они ни были запрошены. Это пример применения *маршрутизации* как механизма эволюционирования. Сервис звездочного рейтинга может существовать в этом состоянии, пока некоторые сервисы все еще будут нуждаться в звездочном рейтинге.

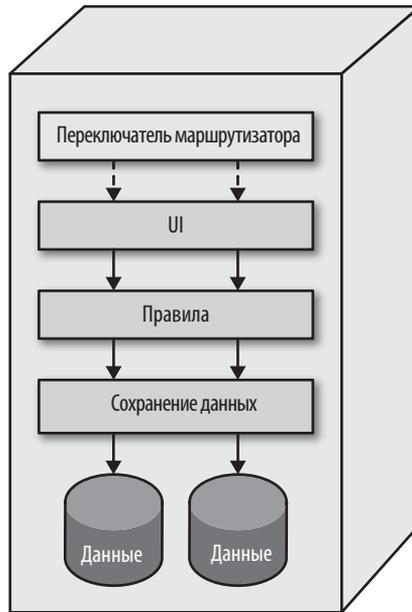


Рис. 6.10. Фаза перехода, в которой сервис StarRating поддерживает оба типа

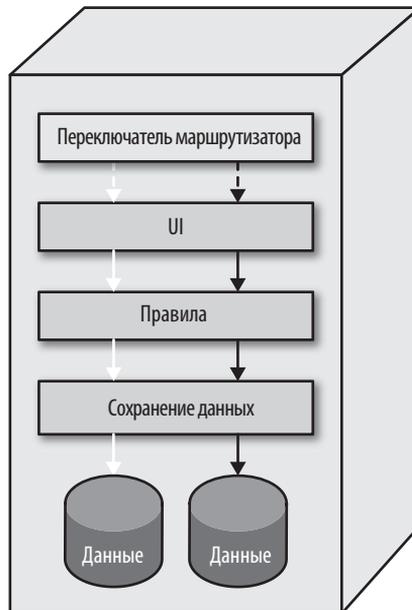


Рис. 6.11. Конечное состояние сервиса StarRating, поддерживающего только новый тип рейтинга

После того как последний сервис зависимости эволюционировал из полнозвездочного рейтинга, разработчики могут удалить директорию старой программы, как показано на рис. 6.11.

Как показано на рис. 6.11, разработчики могут удалить директорию старой программы и, вероятно, удалить прокси-слой для управления разностями версий (или оставить его, чтобы поддерживать дальнейшее эволюционирование).

В этом случае изменение для PenultimateWidgets не будет трудным с точки зрения эволюционирования данных, поскольку разработчики могут сделать дополнительное изменение, позволяющее им добавить в базу данных схему, а не изменять ее. А что можно сказать о таком случае, в котором также должна меняться база данных из-за новой функции? Вернитесь к обсуждению проектирования эволюционирования данных в главе 5.

7

Архитектура с эволюционным развитием: ловушки и антипаттерны

Мы потратили много времени, обсуждая надлежащие уровни связанности в архитектуре. Однако мы также живем в реальном мире и наблюдаем множество связанностей, которые *вредят* способности проекта эволюционировать.

Были выявлены две плохие практики проектирования, которые проявляются в проектах программного обеспечения: *ловушки* и *антипаттерны*. Многие разработчики используют слово *антипаттерны* как жаргонный термин «плохой», но его реальное значение требует уточнения. Антипаттерн программного обеспечения состоит из двух частей. Первая часть: антипаттерн является практикой, которая первоначально кажется хорошей идеей, но оборачивается ошибкой. Вторая часть: для большинства антипаттернов существует множество гораздо лучших альтернатив. Разработчики архитектуры замечают много антипаттернов только в ретроспективе, поэтому их тяжело избежать. *Ловушка* на первый взгляд выглядит как хорошая идея, но немедленно проявляет себя как плохой путь. В этой главе мы рассматриваем ловушки и антипаттерны совместно.

Техническая архитектура

В этом разделе акцент делается на распространенную практику, используемую в промышленности, которая особенно вредит способности команды эволюционировать архитектуру.

Антипаттерн: Vendor King

Некоторые крупные предприятия приобретают программное обеспечение Планирование ресурсов предприятия (ERP — Enterprise Resource Planning) для решения распространенных бизнес-задач, таких как ведение учета, управление инвентаризацией и другие рутинные операции. Это работает, если компании готовы сгибать свои бизнес-процессы и другие решения для того, чтобы приспособить этот инструмент, и может быть использовано стратегически, когда разработчики архитектуры понимают ограничения и преимущества.

Однако многие организации становятся чрезмерно амбициозными в отношении программного обеспечения этой категории, что приводит к *антипаттерну король-поставщик (vendor king)*, архитектура которого целиком строится на основе продукции поставщика, что патологически привязывает организацию к этому инструменту. Компании, приобретающие программное обеспечение поставщика, планируют увеличение пакета с помощью его плагинов, чтобы расширить основную функциональность для приведения ее в соответствие с предметной областью предприятия. Однако продолжительное время инструмент ERP невозможно настроить в достаточной степени для полной реализации того, что необходимо, и разработчики обнаруживают свою беспомощность в результате ограничений инструмента и из-за того, что они сделали его центром архитектурной вселенной. Другими словами, разработчики архитектуры сделали из поставщика короля своей архитектуры, диктующего принятие в будущем решений.

Для того чтобы избежать антипаттерна, следует рассматривать программное обеспечение просто как другую точку интеграции, даже если у него первоначально был широкий круг обязанностей. Предполагая интеграцию на начальном этапе, можно легче менять бесполезные характеристики с другими точками интеграции, свергая короля с престола.

Поместив внешний инструмент или платформу в самое сердце архитектуры, разработчики значительно ограничили свои возможности эволюционировать в двух основных направлениях, а именно технически и с точки зрения бизнес-процесса. Разработчики технически ограничены выбором поставщика в отношении систем хранения данных, поддерживаемой инфраструктурой и массой других ограничений. С точки зрения предметной области крупный инструмент инкапсуляции в конечном счете страдает от антипаттерна «Ловушка на последних 10 %» (с. 206). С точки зрения бизнес-процесса этот инструмент не может поддерживать оптимальным рабочий поток; это — побочный эффект или ловушка на последних 10 %. Большинство компаний завершают работу подчинением платформе, заменяя процессы, а не пытаясь настроить инструмент. Чем больше компаний поступят так, тем меньше отличительных особенностей существует между компаниями, что замечательно, поскольку различие не является преимуществом.

Принцип *давайте остановим работу и назовем это успехом* является одним из тех, которые разработчики обычно учитывают, занимаясь в реальном мире с пакетами ERP. Так как они требуют значительных затрат времени и денежных инвестиций, компании с неохотой на них соглашаются, когда они не работают. Ни один технический отдел не хочет соглашаться на потерю миллионов долларов, а поставщик инструмента не хочет согласиться на плохую многослойную реализацию. Таким образом, каждая из сторон согласна остановить работу и назвать это успехом при большей части нереализованной обещанной функциональности.



Не связывайте свою архитектуру с королем-поставщиком.

Вместо того чтобы стать жертвой антипаттерна короля-поставщика, попробуйте рассматривать продукты поставщика как еще одну точку интеграции. Разработки могут изолировать изменения инструмента поставщика от воздействия их архитектуры, построив между точками интеграции слои противодействия разрушениям.

Ловушка: дырявая абстракция

Все нетривиальные абстракции в какой-то степени дырявые.

— Джоэл Спольски (*Joel Spolsky*)¹

Современное программное обеспечение покоится на башне абстракций: операционные системы, платформы, зависимости и т. п. Как разработчики, мы строим абстракции так, что у нас нет возможности постоянно думать, находясь на нижних уровнях. Если разработчикам потребовалось перевести бинарные цифры, которые поступают с жестких дисков в текст для программы, они никогда не смогут что-нибудь сделать! Одним из триумфов современного программного обеспечения является то, как хорошо мы можем строить эффективные абстракции.

Но абстракции дорого обходятся, потому что нет совершенных абстракций, а если бы они были, то это были бы не абстракции, а нечто реальное. Как считает Джоэл Спольски, все нетривиальные абстракции имеют дырку (протекают). Это является проблемой для разработчиков, потому что мы верим, что абстракции всегда точные, но они часто удивительным образом разрушаются.

Повышенная сложность технологических стеков недавно превратила абстракцию в разрушительную проблему. На рис. 7.1 представлен типичный технологический стек, относящийся примерно к 2005 году.

В этом стеке имя поставщика на блоках меняется в зависимости от местных условий. Со временем, по мере того как программное обеспечение становится все в большей степени специализированным, наш технологический стек становится все более сложным, как показано на рис. 7.2.

¹ <http://russian.joelonsoftware.com/Articles/LeakyAbstractions.html>

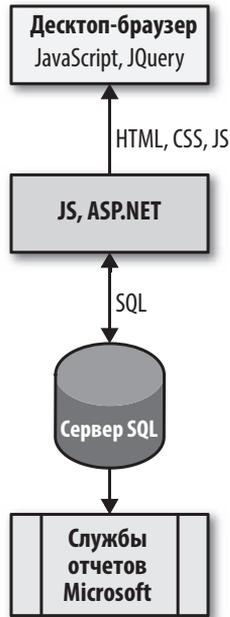


Рис. 7.1. Типичный технологический стек 2005 г.

Как видно на рис. 7.2, каждая из частей экосистемы программного обеспечения расширилась и стала более сложной. По мере того как проблемы, с которыми сталкиваются разработчики, становятся все более сложными, такими же сложными становятся и их решения.

Первоначальная дырявая абстракция, где разрушающая абстракция на низком уровне приводит к неожиданному хаосу, является одним из побочных эффектов повышения сложности технологического стека. Что, если одна из абстракций на самом нижнем уровне проявляет сбой, например, некоторый неожиданный побочный эффект из кажущегося безвредным вызова базы данных? Поскольку существует так много слоев, то этот сбой будет двигаться в верхнюю часть этого стека, возможно, вызывая на своем пути «метастазы», проявляясь в глубоко встроенном сообщении об ошибке в UI. Отладка и ретроспективный анализ становятся тем затруднительнее, чем сложнее технологический стек.

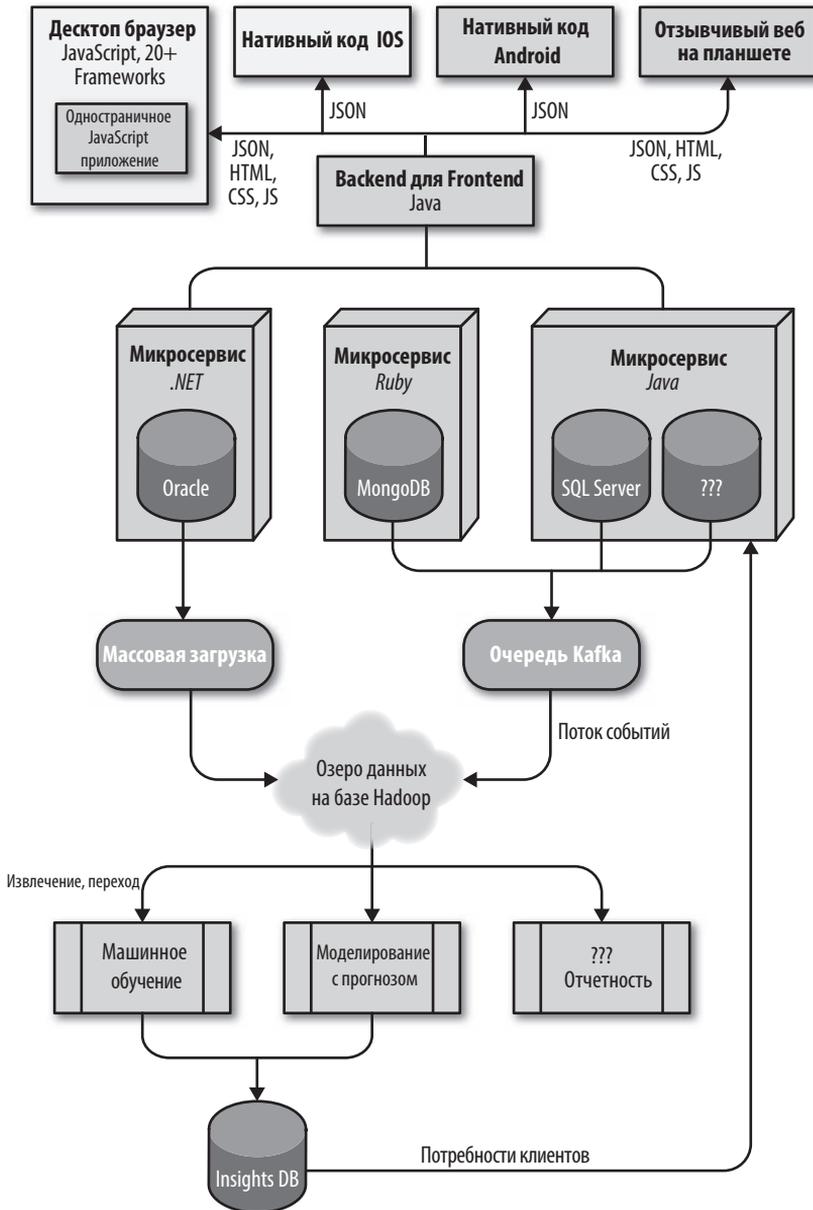


Рис. 7.2. Типичный программно-реализованный стек, относящийся к 2016 году, с множеством движущихся частей

Старайтесь до конца понять по крайней мере один уровень абстракции, находящийся ниже того уровня, на котором вы обычно работаете.

— *Мудрецы программного обеспечения*

Несмотря на то что понимание слоя ниже является хорошим советом, ему все сложнее следовать по мере специализации и усложнения программного обеспечения.

Повышение сложности технологического стека является примером задачи динамического равновесия. Не только меняется экосистема, но и составляющие ее части становятся со временем более сложными и перепутанными. Предложенный механизм защиты эволюционирующих изменений, а именно применение функций пригодности, может защитить хрупкие точки соединений архитектуры. Архитекторы задают инварианты в ключевых точках интеграции, такие как функции пригодности, которые работают как часть конвейера развертывания, гарантируя, что абстракция не начнет протекать нежелательным образом.



Постарайтесь выявить хрупкие места внутри своего сложного технологического стека и применить автоматическую защиту с помощью функций пригодности.

Антипаттерн: ловушка на последних 10 %

Другой тип часто возникающей ловушки существует на другом конце спектра абстракций, с пакетами программного обеспечения, платформами и фреймворками.

Одно время Нил был руководителем технического отдела консалтинговой фирмы, которая выполняла для клиентов проекты, включая Microsoft Access, на различных языках четвертого поколения (4GL). В итоге Нил помог принять решение об устранении Access, и в конце

концов из предметной области были исключены все 4GL, после того как он установил, что каждый проект Access начинался с успехом, а заканчивался отказом. Нилу хотелось установить почему. Он и его коллеги наблюдали, что в Access и других 4GL, популярных в то время, 80 % того, что хотел клиент, было быстро и легко построить. Эти среды моделировались как инструменты быстрой разработки приложений, с поддержкой перетаскивания для пользовательских интерфейсов и т. п. Однако остальные 10 % того, что хотел клиент, были чрезвычайно сложными, если вообще выполнимыми, потому что эта функциональность не была встроена в инструмент, фреймворк или язык. Поэтому умные разработчики разработали способ взломать инструменты, чтобы заставить все работать: добавили скрипт для выполнения, где ожидалось статические вещи, цепные методы и другие «костыли». Взлом дает вам только 80–90 %. В конечном итоге инструмент не может полностью решить проблему. Фраза, которую мы ввели в обращение, — *ловушка на последних 10 %* — означает разочарование, которое оставляет каждый проект. Несмотря на то что языки 4GL упрощают ускоренное строительство простых элементов, они не масштабируются, чтобы выполнить требования реального мира. Разработчики вернулись к языкам общего назначения.

ПРОЕКТ ИВМ САН-ФРАНЦИСКО

В конце 1990-х годов ИВМ выступила с амбициозным планом написать программу, которая решит все задачи бизнеса, и писать больше ничего не придется. Команда разработчиков приступила к проектированию набора повторно используемых бизнес-компонентов, написанных на языке того поколения Java, которое могло инкапсулировать всю бизнес-функциональность в широкие категории: книгу учета бухгалтерии, товарные запасы, продажи и т. п. В какой-то момент компания ИВМ заявила, что этот проект представляет собой крупнейший Java-проект на Земле (<http://www.drdoobs.com/ibms-san-francisco-project/184415597>). Проект предоставил первые несколько модулей ядра, и разработчики начали использовать фреймворк, который привел к его кончине. Многие функции оказались ненужными, а много важных функций при этом отсутствовали.

Проект Сан-Франциско демонстрирует завышенную самооценку разработчиков, которые пытались следовать своим инстинктам для категоризации и классификации всего на свете. Некоторые беспорядочные события реального мира нарушают изящные решения, включая все бизнес-процессы!

Проект Сан-Франциско, в итоге, провалился, потому что его разработчики постепенно осознали отрезвляющий факт — неважно, насколько сильно старались разработчики, потому что они никогда не смогут очистить все до достаточно детализированных свойств, часть проблемы *бесконечного регресса*: серии предложений, которые продолжают опираться на другие предложения, идущие в бесконечность. В программном обеспечении постоянный регресс проявляется как попытка уточнить все на свете на конечном уровне детализации, но всегда есть другой уровень детализации, находящийся ниже любой существующей детали.

Антипаттерн: неправильное повторное использование кода

Как отрасль, мы получили значительные преимущества от повторного использования фреймворков и сборок библиотек сторонних разработчиков, часто с открытым исходным кодом и свободно распространяемых. Понятно, что возможность повторного использования кода — это преимущество. Однако, как это часто бывает в случае хороших идей, некоторые компании неправильно используют эту возможность и создают для себя проблемы. Каждой корпорации хотелось бы повторно использовать код, потому что программное обеспечение выглядит также модульно, как и электронные компоненты. Однако, несмотря на обещание, что существует действительно модульное программное обеспечение, это обещание постоянно обходит нас.

Повторное использование программного обеспечения больше напоминает трансплантацию органа, чем складывание вместе кубиков Лего.

— Джон Д. Кук (John D. Cook)

Несмотря на то что разработчики языков уже давно обещают разработчикам программного обеспечения кубики Лего, нам все еще кажется, что приходится иметь дело с органами. Повторное применение программного обеспечения затруднительно и не происходит автоматически. Многие оптимистично настроенные менеджеры полагают, что любой код, который написали разработчики, по своей природе предназначен для повторного использования, но это далеко не всегда так. Многие компании предпринимали попытки и преуспевали в написании действительно способного к повторному использованию кода, но это оказалось сложным и требующим значительного мыслительного напряжения процессом. Разработчики часто тратят много времени, пытаясь собрать повторно используемые модули, которые, как оказывается, мало используются повторно.

В сервис-ориентированной архитектуре была распространена практика находить общность и использовать ее как можно больше повторно. Например, представьте, что у компании есть два контекста: **Checkout** и **Shipping**. В сервис-ориентированной архитектуре разработчики архитектуры обнаружили, что оба контекста включают концепцию **Customer**. Это, в свою очередь, стимулировало их объединить обоих покупателей в единый сервис **Customer**, связав **Checkout** и **Shipping** в совместно используемый сервис. Архитекторы работали для достижения цели окончательной *каноничности* (canonicality) в сервис-ориентированной архитектуре — все до одной концепции имеют одно (общее) исходное положение.

По иронии, чем больше усилий разработчики вкладывали в возможность повторного использования кода, тем тяжелее им было пользоваться. Процедура обеспечения возможности повторного использования кода содержит дополнительные опции и точки принятия решений для обеспечения возможности различного использования. Чем больше разработчики добавляют средств для повторного использования, тем больший вред они наносят основному *юзабилити* кода.



Чем больше у кода возможностей для повторного применения, тем менее пригодным к применению он становится.

Другими словами, простота использования кода часто обратно пропорциональна возможности его повторного применения. Когда разработчики пишут код, предназначенный для повторного использования, они должны добавить элементы для обеспечения множества способов его фактического использования разработчиками. Вся эта последующая правка затрудняет для разработчиков использование этого кода по одному из ее назначений.

Микросервисы избегают повторного использования кода, применяя философию *предпочтительнее дублирование, а не связанность*: повторное использование предполагает связанность, а архитектура микросервисов в значительной степени лишена связанности. Однако целью микросервисов является не принять дублирование, а скорее изолировать объекты внутри областей. Сервисы, совместно использующие общий класс, не являются больше независимыми. В архитектуре микросервисов `Checkout` и `Shipping` имеют собственные внутренние представители `Customer`. Если им необходимо сотрудничать по информации, относящейся к клиенту, они направляют соответствующую информацию друг другу. Архитекторы не пытаются примирять и объединять несопоставимые версии `Customer` в своей архитектуре. Эти преимущества повторного использования являются иллюзорными, и вносимая ими связанность вносит определенные недостатки. Поэтому, несмотря на то что разработчики понимают недостатки дублирования, они компенсируют этот локализованный ущерб слишком большой вносимой связанностью.

Повторное использование кода может быть определенной пользой, но также и потенциальным обязательством. Следует убедиться, что соединительные точки, вносимые в код, не конфликтуют с остальными целями архитектуры. Например, архитектуры микросервисов обычно используют шаблоны сервисов (описание приведено в разделе «Практический пример: шаблоны сервисов» на с. 185) для соединения сервисов вместе, что помогает привести к одному виду определенные проблемы архитектуры, такие как мониторинг или сбор данных.

Практический пример: принцип повторного использования в PenultimateWidgets

PenultimateWidgets имеет очень специфические требования для ввода данных в специализированной сетке для их администрирования. Поскольку приложение требовало этого представления в нескольких местах, PenultimateWidgets решили создать повторно используемый компонент, включающий пользовательский интерфейс, проверку и другие полезные функции по умолчанию. Используя этот компонент, разработчики могут легко создавать новые, богатые интерфейсы администрирования.

Однако фактически ни одно архитектурное решение не проходило без компромиссов. Со временем команда, занимающаяся этим компонентом, стала подразделением организации, связав несколько лучших разработчиков PenultimateWidgets. Другие команды, использующие этот компонент, должны запросить новые функции у команды, занимающейся компонентом, что привело к потоку запросов на отладку. Хуже всего, что лежащий в основе код не соответствовал современным веб-стандартам, затрудняя новую функциональность или делая ее невозможной.

Несмотря на то что разработчики PenultimateWidgets добились повторного использования, это в конечном счете привело к эффекту «бутылочного горлышка». Одно из преимуществ повторного использования заключается в том, что разработчики могут быстро создавать новые вещи. Тем не менее, если команда не сможет идти в ногу с темпами инновационного динамического равновесия, повторное использование компонента технической архитектуры обречено на окончательное превращение в антипаттерн.

Мы не предлагаем командам избегать разработки повторно используемых средств, но при этом предлагаем постоянно их оценивать для гарантии того, что они все еще приносят пользу. В случае с PenultimateWidgets, после того как разработчики осознали, что этот компонент был «бутылочным горлышком», они убрали соединитель-

ные точки. Любая команда, которая хотела бы продублировать код компонента, чтобы добавить собственные функции, может это сделать (пока команда разработки приложения поддерживает изменения), а любая команда, которой не хотелось бы использовать новый метод, полностью освобождается от старого кода.

Из опыта компании PenultimateWidgets:



Когда соединительные точки препятствуют эволюционированию или наносят вред другим важным характеристикам архитектуры, эти соединения следует разорвать с помощью ветвления или дублирования.

В случае PenultimateWidgets они разорвали соединение, разрешив командам самостоятельно использовать общий код, став его владельцами. Несмотря на дополнительную нагрузку это избавило их от необходимости добавлять новые функции. В других случаях, возможно, некоторый совместно используемый код можно отделить от более крупной части программного обеспечения, предоставив возможность использовать более селективное соединение и постепенное устранение этого соединения.



Разработчики архитектуры должны непрерывно оценивать пригодность различных возможностей архитектуры для гарантии того, что они все еще приносят пользу и еще не превратились в антипаттерны.

Слишком часто архитекторы принимают решение, которое в момент принятия кажется им правильным, но со временем становится неудачным из-за изменения условий, таких как динамическое равновесие. Например, архитекторы проектируют систему как настольное приложение, но отрасль относит их в сторону веб-приложения по мере изменения привычек пользователей. Первоначальное решение не было неверным, но экосистема неожиданно изменилась.

Ловушка: разработки ради резюме

Архитекторы в восторге от захватывающих новых разработок в экосистеме разработки программного обеспечения и хотят играть с новейшими игрушками. Однако чтобы выбрать эффективную архитектуру, они должны внимательно изучить предметную область и выбрать наиболее подходящую архитектуру, которая обеспечивает желаемые возможности с наименьшими вредными ограничениями. За исключением, конечно, того случая, когда целью разработки архитектуры является *разработка ради резюме*. Разработчики используют каждый фреймворк и библиотеки, чтобы рекламировать эти знания в своем резюме.



Не стройте архитектуру ради архитектуры — ведь вы пытаетесь решить задачу.

Всегда необходимо понять предметную область, прежде чем выбирать архитектуру, а не другой окружной путь.

Инкрементные изменения

Многие факторы в разработке программного обеспечения затрудняют инкрементные изменения. Несколько десятилетий программное обеспечение писалось не ради обеспечения динамичности, а скорее ради снижения затрат, совместного использования ресурсов и других внешних ограничений. Следовательно, многие организации не имели строительных блоков для оказания поддержки эволюционирования архитектуры.

Как обсуждалось в книге *Непрерывная поставка*, многие современные практики проектирования поддерживают эволюционирующие архитектуры.

Антипаттерн: ненадлежащее управление

Архитектура программного обеспечения никогда не существует в вакууме; часто она является отражением среды, в которой была разработана. Десять лет назад операционные системы были дорогостоящим коммерческим предложением. Аналогичным образом, серверы баз данных, серверы приложений и вся инфраструктура для размещаемых приложений была коммерческой и дорогостоящей. Разработчики реагировали на это оказываемое реальным миром давление, разрабатывая архитектуры для обеспечения максимально возможного совместного использования ресурсов. Многие паттерны архитектур, такие как SOA, расцвели в эту эпоху. В этой среде появилась модель общего управления для обеспечения максимально возможного совместного использования ресурсов как мера сокращения расходов. Многие коммерческие мотивации для инструмента, такие как серверы приложений, выросли из этой тенденции. Однако упаковка многочисленных ресурсов в машинах нежелательна с точки зрения разработки из-за случайной связанности. Неважно, насколько хороша изоляция между совместно используемыми ресурсами, состязание за обладание ресурсами в конечном счете даст о себе знать.

В течение последнего десятилетия появились изменения в динамическом равновесии экосистемы разработки. Теперь разработчики могут построить архитектуру, в которой компоненты имеют высокую степень изоляции (как в микросервисах), устраняя случайную связанность, усложняемую совместно используемой средой. Но многие компании все еще придерживаются схемы старого управления. Модель управления, которая оценивает совместно используемые ресурсы и гомогенизированную среду, не имеет большого смысла из-за недавних достижений, таких как движение DevOps.

Каждая компания — теперь софтверная компания.

— *Журнал Forbes, 30 ноября 2011 г.*

В своей знаменитой фразе Forbes имел в виду, что если приложение авиакомпании для iPad ужасно — это в итоге повлияет на чистую прибыль компании. Для любой передовой компании необходима компетентность в вопросах программного обеспечения, и эта необходимость повышается для любой компании, которой хотелось бы сохранить конкурентоспособность. Эта компетентность включает в себя то, как компания управляет активом разработки, таким как среда.

Когда разработчики могут бесплатно создать такие ресурсы, как виртуальные машины и контейнеры (монетарные или контейнеры времени), модель управления, которая оценивает однозначное решение, становится *ненадлежащим управлением*. Для среды микросервисов появился лучший подход. Одной из обычных характеристик микросервисов является включение многоязычных сред, в которых каждая команда сервисов может выбрать подходящий технологический стек для реализации своего сервиса, вместо того чтобы пытаться усреднить все по стандарту предприятия. Традиционные разработчики предприятий морщатся, когда слышат этот совет, потому что он диаметрально противоположен традиционному подходу. Однако цель большинства проектов микросервисов заключается не в том, чтобы непринужденно подбирать разные технологии, а скорее в выборе соответствующей технологии по размеру проблемы.

В современных средах это является ненадлежащим управлением для усреднения по единичному технологическому стеку, так как ведет к неадекватным усложнениям проблемы, для которой управленческие решения повышают бесполезные усилия для получения решения. Например, стандартизация по одной реляционной базе данных поставщика является обычной практикой на больших предприятиях, что связано со следующими причинами: согласование по всем проектам, легко заменяемый персонал и т. п. Однако побочным эффектом этого метода является то, что большинство проектов страдают от чрезмерного технического усложнения системы. Когда разработчики создают монолитные архитектуры, выбор управления

влияет на всех. Поэтому при выборе базы данных архитектор должен учитывать требования каждого проекта, который будет пользоваться этой возможностью, и делать выбор, в дальнейшем используемый в наиболее сложных случаях. К сожалению, многие проекты не будут иметь наиболее сложный случай или что-то в этом роде. Небольшой проект может иметь простую систему хранения данных, и все же для согласованности требуется сложный сервер баз данных промышленного стандарта.

В случае микросервисов, поскольку ни один из сервисов не связан через техническую архитектуру или архитектуру данных, различные команды могут выбирать надлежащий уровень сложности, требуемый для реализации этого сервиса. Окончательной целью является упрощение, необходимое для настройки комплекса стека сервисов по техническим требованиям. Разделение на части работает лучше, когда команда полностью владеет своими сервисами, включая аспекты работы.

ПРИНУДИТЕЛЬНОЕ УМЕНЬШЕНИЕ СВЯЗАННОСТИ

Одной из целей архитектуры микросервисов является сильное уменьшение связанности технической архитектуры, что позволяет заменять сервис без побочных эффектов. Однако если все разработчики совместно используют один и тот же код или даже платформу, *отсутствие* связанности требует от разработчиков некоторой дисциплины (поскольку велико искушение повторного использования существующего кода) и средств защиты, чтобы гарантировать, что связанность не возникнет случайно. Построение сервисов в различных технологических стеках является одним из способов достижения несвязанности технической архитектуры. Многие компании стараются избегать использования этого метода, так как опасаются, что он будет препятствовать перемещению персонала по проектам. Однако Чед Фаулер¹ (Chad Fowler), разработчик компании Wunderlist, попробовал применить противо-

¹ Фаулер Ч. Программист-фанатик. — СПб.: Питер, 2018. — 208 с.: ил. — (Серия «Библиотека программиста»).

ложный подход: он *настоял* на том, чтобы команды использовали другие технологические стеки для исключения несоответствующей связанности. Его принцип состоит в том, что возникающая случайно связанность является большей проблемой, чем мобильность разработчика.

Многие компании инкапсулируют отдельную функциональность в платформу как услугу¹ для внутреннего применения, скрывая выборы технологии (и тем самым возможности связанности) за хорошо определенный интерфейс.

С точки зрения практического управления в крупных организациях оказалось, что модель *управление «золотой середины»* (Goldilocks Governance) работает хорошо: для стандартизации выбираются три технологических стека (простой, промежуточный и сложный) и предоставляется возможность отдельным требованиям сервиса управлять требованиями стека. Это дает командам свободу выбора подходящего технологического стека и одновременно дает компании некоторые преимущества стандартов.

Практический пример: модель управления «золотой середины» в PenultimateWidgets

В течение многих лет архитекторы PenultimateWidgets пытались стандартизировать все разработки на Java и Oracle. Однако по мере того, как они создавали более детализированные сервисы, они осознали, что стек привел к появлению в небольших сервисах значительной сложности. Но разработчикам не хотелось полностью использовать подход «каждый проект выбирает свой собственный технологический стек» для микросервисов, им хотелось сохранить возможность перенесения знаний и навыков из одного проекта в другой. В конце концов они выбрали модель управления «золотой середины» с тремя технологическими стеками:

¹ https://ru.wikipedia.org/wiki/Платформа_как_услуга

Небольшой

Для простых проектов без строгой масштабируемости или требований к производительности они выбрали Ruby on Rails и MySQL.

Средний

Для средних проектов они выбрали GoLang и один из Cassandra, MongoDB или MySQL в качестве бекэнда, в зависимости от требования к данным.

Большой

Для крупных проектов они сохранили Java и Oracle, поскольку они хорошо работают с переменными характеристиками архитектуры.

Ловушка: недостаточная скорость для релиза

Практика проектирования в случае непрерывной поставки рассматривает факторы, которые замедляют релизы программного обеспечения, и эту практику следует считать атомарной для эволюционирующей архитектуры, чтобы она была успешной. Несмотря на то что использование непрерывной поставки не требуется для эволюционирующей архитектуры, между возможностью выпускать программное обеспечение и способностью эволюционировать существует сильная корреляция.

Если компания строит культуру проектирования вокруг непрерывного развертывания, ожидая, что все изменения проложат свой путь в производство, после того как они прошли конвейер развертывания, разработчики привыкают к постоянным изменениям. С другой стороны, если релизы являются формальным процессом, который требует большого объема специализированных работ, шансы достичь эволюционирования архитектуры уменьшаются.

Непрерывная поставка стремится к результатам на основе данных и использует показатели, чтобы научиться оптимизировать проект.

Разработчики должны быть способны измерять разные параметры, чтобы понять, как их улучшить. Одним из ключевых показателей, отслеживаемых при непрерывной поставке, является *время цикла* (cycle time), который относится к *времени внедрения* (lead time): время между возникновением идеи и временем, когда идея проявится в работающем программном обеспечении. Однако время цикла включает много субъективных активностей, таких как оценка, расстановка приоритетов и т. п., что превращает это время в непригодный показатель для проектирования. Вместо этого параметра непрерывное развертывание отслеживает *время цикла*: время, прошедшее между началом и завершением единицы работы, которая в данном случае является разработкой программного обеспечения. Отсчет времени цикла начинается, когда разработчик начинает работу над новой функцией, и завершается, когда эта функция выполняется в производственной среде. Цель этого показателя — измерить эффективность проектирования; уменьшение времени цикла является одной из ключевых целей непрерывной поставки.

Время цикла также имеет решающее значение для эволюционирующей архитектуры. В биологии плодовые мушки обычно используются в экспериментах для демонстрации генетических характеристик отчасти потому, что они имеют короткое время цикла — новые поколения появляются достаточно быстро, чтобы увидеть ощутимые результаты. Это же справедливо в эволюционирующей архитектуре — короткое время цикла означает, что архитектура может быстрее эволюционировать. Поэтому время цикла проекта определяет то, как быстро архитектура может эволюционировать. Другими словами, скорость эволюционирования пропорциональна времени цикла, что можно выразить следующим образом:

$$v \propto c,$$

где v обозначает скорость изменения, а c — время цикла. Разработчики не могут эволюционировать систему быстрее, чем время цикла проекта. Другими словами, чем быстрее команды могут выпустить программное обеспечение, тем быстрее могут эволюционировать части этой системы.

Поэтому время цикла является важным показателем в проектах с эволюционирующей архитектурой — короткое время цикла подразумевает возможность быстрее эволюционировать. Фактически, время цикла превосходно могло бы подойти для атомарной, основанной на процессе, функции пригодности. Например, разработчики подготовили проект с конвейером развертывания и с автоматизацией, добившись времени цикла в три часа. Со временем время цикла постепенно увеличивалось по мере того, как разработчики стали добавлять в конвейер развертывания больше проверок и точек интеграции. Поскольку время выхода на рынок является важным показателем этого проекта, они установили функцию пригодности, чтобы сигнализировать, когда время цикла превышает четыре часа. После того как был достигнут порог, разработчики могли принять решение реконструировать работу конвейера развертывания или прийти к выводу, что время цикла, равное 4 часам, вполне приемлемо. Функции пригодности могут отображать любые характеристики, которые разработчик хотел бы контролировать в проекте, включая показатели проекта. Объединение функциональностей проекта в виде функций пригодности дает возможность разработчикам установить точки принятия последующих решений (эта процедура известна как *последний ответственный момент*) для повторной проверки решений. В предыдущем примере разработчики должны теперь принять решение, что важнее: трехчасовой цикл или набор тестов, которые нужно пройти на месте. В большинстве проектов разработчики принимают это решение неявным образом, не обращая внимания на постепенно растущее время цикла и, таким образом, никогда не отдавая предпочтения конфликтующим целям. С функциями пригодности они могут установить пороги вокруг точек ожидаемых в будущем решений.



Скорость эволюционирования зависит от времени цикла; короткий цикл допускает более быстрое эволюционирование.

Хорошее проектирование, развертывание и практики релиза программного обеспечения имеют большое значение для успешного

эволюционирования архитектуры, что, в свою очередь, дает новые возможности для развития бизнеса с помощью разработки на основе гипотезы.

Проблемы бизнеса

И наконец, мы поговорим о ненадлежащей связанности, основанной на проблемах бизнеса. В большинстве случаев представители бизнеса — это не гнусные персонажи, пытающиеся осложнить жизнь разработчикам. У них просто есть приоритеты, приводящие к несоответствующим решениям с архитектурной точки зрения, которые непреднамеренно ограничивают будущие варианты. Мы рассмотрим несколько ловушек и антипаттернов в области бизнеса.

Ловушка: адаптация продукта

Продавцы хотят иметь возможности продавать. Карикатурой на продавца является человек, продающий любую требуемую функцию до того, как определить, содержит ли продукт эту функцию. Таким образом, продавцы хотят бесконечно адаптировать программное обеспечение, чтобы его продать. Однако эти возможности обернутся определенными затратами при использовании метода реализации.

Уникальная сборка для каждого заказчика

В этом сценарии продавцы обещают уникальные версии функций за короткое время, вынуждая разработчиков использовать такие методы, как контроль версий и разметка для отслеживания версий.

Постоянные переключения функций

В главе 3 мы описали переключатели функций, которые иногда стратегически используются для создания постоянной адаптации. Разработчики могут использовать переключатель функций для создания разных версий под различных клиентов или для создания «условно-бесплатной» версии продукта, то есть бес-

платной версии, когда пользователям за использование премиум-функций придется вносить плату.

Адаптация на основе продукта

Некоторые продукты зашли так далеко, что позволяют выполнить адаптацию через пользовательский интерфейс. Функции в этом случае являются постоянными частями приложения и требуют такого же внимания, как и все остальные функции продукта.

С такими дополнениями, как переключение функций и адаптация, время тестирования значительно увеличивается, поскольку этот продукт содержит много перестановок возможных путей. Наряду со сценариями тестирования, число функций пригодности, необходимых разработчику для его работы, может увеличиться для защиты возможных перестановок.

Адаптация также препятствует развитию, но это не должно препятствовать компаниям создавать настраиваемое программное обеспечение, а должно способствовать реалистично оценивать связанные с этим затраты.

Антипаттерн: составление отчетов

В зависимости от предметной области, большинство приложений имеют разные области применения. Например, некоторым пользователям нужна запись о заказе, в то время как другие требуют отчетов для анализа. Организации борются за предоставление всех возможных перспектив (например, ввод заказов и ежемесячные отчеты), которые требуют предприятия, особенно если все должно происходить из одной и той же монолитной архитектуры и/или структуры базы данных. В эпоху сервис-ориентированной архитектуры архитекторы пытались поддержать каждую бизнес-проблему с помощью одного и того же набора «повторно используемых» сервисов. Они обнаружили, что чем более общим является сервис, тем больше разработчикам необходимо адаптировать его для использования.

Предоставление отчетности является хорошим примером неадекватной связанности в монолитной архитектуре. Разработчики и администраторы баз данных хотят использовать схему базы данных для системы записи и отчетности, но сталкиваются с проблемами, так как проектирование для поддержки того и другого не оптимально ни для того, ни для другого. Распространенная ловушка, которую разработчики и проектировщики системы подготовки отчетов стоворились создать в слоистой архитектуре, иллюстрирует напряженность между областями. Разработчики создают слоистую архитектуру, чтобы разорвать случайное соединение, создавая слои изоляции и разделения областей. Однако в сервисах подготовки отчетов не нужно разделять слои для поддержки их функционирования — нужны только данные. Кроме того, маршрутизация запрашивает дополнительное время поддержки для прохождения через слои. Поэтому многие организации с хорошей слоистой архитектурой позволяют проектировщикам системы составления отчетов напрямую присоединять отчеты к схеме базы данных, устраняя возможность вносить изменения в схему, не разрушая отчеты. Это хороший пример конфликтующих целей бизнеса, разрушающих работу разработчиков и чрезвычайно затрудняющих эволюционные изменения. Несмотря на то, что никто не ставил перед собой задачу сделать так, чтобы системе было тяжело эволюционировать, это был совокупный результат принятых решений.

Многие архитектуры микросервисов решили задачу предоставления отчетов, разделив характеристики, при этом изоляция сервисов способствовала разделению, а не консолидации. Архитекторы обычно создают эти архитектуры, используя поток событий или очередь сообщений для заселения области «система составления отчетов» в базе данных, каждая из которых встроена внутри архитектурного кванта сервиса с использованием окончательного согласования, а не характеристики транзакций. Набор сервисов отчетности прислушивается также к потоку событий, заселяя денормализованную базу данных отчетов, оптимизированную для составления отчетов. Использование окончательной согласованности освобождает разработчиков от координации, являющейся с точки зрения архитектуры формой связи, позволяя использовать разным пользователям разные абстракции этого приложения.

Например, в архитектуре микросервисов компании PenultimateWidgets использовались области, разделенные на ограниченные контексты, каждый из которых обладал данными для «системы составления отчетов» этой области. Разработчики компании PenultimateWidgets использовали окончательную согласованность и очередь сообщений для заселения и связи, а также набор сервисов отчетности, отделенных от области сервисов, как показано на рис. 7.3.

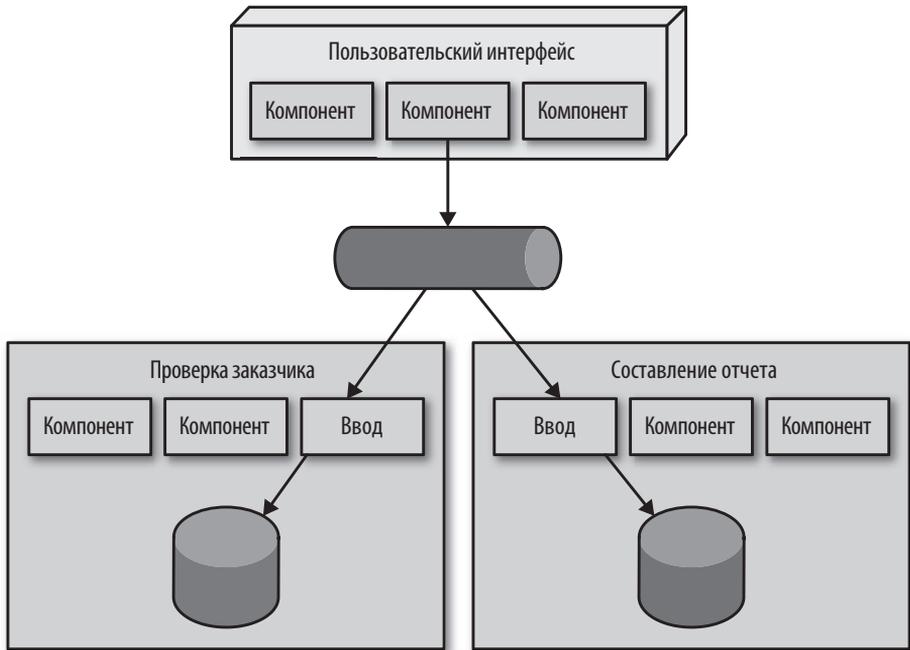


Рис. 7.3. Разделение области и сервисов отчетности в компании PenultimateWidgets, согласованных с помощью очереди сообщений

Как видно на рис. 7.3, когда UI сообщает об операции Создать, Прочсть, Обновить, Удалить (CRUD – Create, Read, Update, Delete), микросервис области и сервис отчетности прислушиваются к уведомлениям и выполняют соответствующие действия. Таким образом, набор сервисов отчетности управляет уведомлением о нарушениях, не влияя на сервис области. Удаление ненадлежащей связанности, внесенной объединенными областями и составлением отчетов, дает

возможность каждой команде сосредоточиться на специфических и в то же время более простых задачах.

Ловушка: горизонты планирования

Процессы составления бюджета и планирования часто приводят к необходимости делать предположения и принимать заблаговременные решения, служащие основанием для этих предположений. Однако более крупный горизонт планирования без возможности его повторного рассмотрения означает, что решения (или предположения) сделаны в условиях минимального объема информации. На этапе раннего планирования разработчики тратят значительные усилия на такие действия, как исследования, часто в форме чтения соответствующих материалов, чтобы проверить сделанные предположения. На основании своих исследований, какая практика наилучшая или что является лучшим в этом классе на момент проведения исследования, формируется часть основополагающих предположений, прежде чем разработчики пишут любой код или выпускают программное обеспечение для конечных пользователей. Большой объем усилий, который тратится на предположения, даже если они через полгода оказываются ошибочными, приводит к сильной привязанности к ним. Невозвратные затраты¹ описывают решения, принятые под влиянием эмоциональной вовлеченности. Проще говоря, чем больше кто-то тратит на что-то времени или усилий, тем тяжелее становится от них отказаться. В случае программного обеспечения это проявляется в форме *иррациональной привязанности к артефактам* — чем больше времени и усилий вы тратите на планирование или на документ, тем вероятнее вы будете защищать то, что содержится в составленном плане или документе, даже перед лицом доказательств того, что они неточные или устаревшие.



Не становитесь иррационально привязанными к сделанным вами артефактам.

¹ https://ru.wikipedia.org/wiki/Невозвратные_затраты

Остерегайтесь продолжительных циклов планирования, которые вынуждают разработчиков принимать необратимые решения и находить способы воздерживаться от окончательного решения. Разделяйте большую программу работ на небольшие части, проводите заблаговременные тесты проектной документации на реализуемость выбранных вариантов архитектуры и инфраструктуры разработки. Архитекторы должны избегать технологий, которые требуют значительных начальных инвестиций до того, как программное обеспечение фактически будет разработано (например, крупные контракты по лицензиям и обслуживанию). Архитекторы должны подтвердить обратную связь с конечным пользователем, и то, что было выполнено тестирование фактической пригодности предлагаемой технологии для задачи, которую они пытаются решить.

8

Внедрение эволюционной архитектуры

И наконец, мы рассмотрим те шаги, которые необходимы для реализации идей по эволюционирующей архитектуре. Они включают в себя технические проблемы и проблемы предметной области, а также влияние организации и команды. Мы также порекомендуем, с чего начинать и как продавать эти идеи.

Организационные факторы

Влияние архитектуры программного обеспечения обладает удивительной широтой по различным факторам, обычно не связанным с программным обеспечением, включая влияние команды, составление бюджета и другое.

Команды, образующиеся вокруг областей, а не технических возможностей, имеют ряд преимуществ, когда дело касается эволюционирующей архитектуры. Они проявляют некоторые общие характеристики.

Кросс-функциональные команды

Предметно-центрированные команды стремятся стать *кросс-функциональными*, где каждая роль проекта охватывается кем-то из проекта. Целью предметно-центрированных команд является устранение трений, возникающих при работе. Другими словами, команда

исполняет все роли, необходимые для проектирования, внедрения и развертывания своего сервиса, включая традиционно отдельные роли, такие как выполнение работ. Но эти роли должны меняться, чтобы приспособлять новую структуру, которая выполняет следующие функции:

Бизнес-аналитика

Должна координировать цели этого сервиса с остальными сервисами, включая координацию с другими командами.

Архитектура

Проектирование архитектуры для исключения надлежащей связанности, которая усложняет инкрементные изменения. Обратите внимание, что для этого не требуется экзотическая архитектура, такая как архитектура микросервисов. Хорошо спроектированное модульное монолитное приложение может проявлять такую же способность обеспечивать инкрементные изменения (хотя разработчики должны явным образом проектировать приложения для поддержки этого уровня изменений).

Тестирование

Тестирования должны стать привычными для задач, возникающих с интеграцией различных областей, таких как среда интеграции, создание и поддержка контрактов и т. п.

Выполнение работ

Разделение на части сервиса и развертывание их по отдельности (часто параллельно с существующими сервисами и непрерывно развертываемыми сервисами) является для многих организаций с традиционными ИТ-структурами ужасной проблемой. Разработчики примитивных архитектур старой школы убеждены, что компонентная и операционная модульность является одним и тем же, но в реальном мире это часто не так.

Автоматизация задач DevOps, таких как инициализация машины и развертывание, имеют решающее значение для успеха.

Данные

Администраторы баз данных должны иметь дело с новой гранулярностью, транзакцией и системой регистрации проблем.

Одной из целей кросс-функциональных команд является устранение трений, возникающих при координации. В случае традиционных разрозненных команд разработчики часто должны дожидаться, когда администратор баз данных внесет изменения, или ждать того, кто после завершения своей работы освободит ресурсы. Если сделать все роли локальными, это позволит устранить случайное возникновение трений при координации работы разрозненных команд.

Несмотря на то что очень здорово иметь для исполнения всех ролей квалифицированных инженеров в каждом проекте, большинство компаний не считают это удачным вариантом. Области ключевых навыков всегда ограничены внешними силами, такими как требования рынка. Поэтому многие компании стремятся создать кросс-функциональные команды, но не могут себе этого позволить из-за ограниченности ресурсов. В таких случаях ограниченные ресурсы могут использоваться совместно несколькими проектами. Например, вместо того чтобы иметь одного инженера по эксплуатации в каждом сервисе, их можно перемещать между несколькими различными командами.

При моделировании архитектуры и команды вокруг предметной области общая единица изменений теперь обрабатывается в одной команде, что снижает искусственное трение. Предметно-центрированная архитектура может все еще использовать слоистую архитектуру для других преимуществ, таких как разделение обязанностей. Например, использование определенного микросервиса может зависеть от фреймворка, который реализует слоистую архитектуру, позволяя команде легко заменить тот или иной технический слой. Микросервисы инкапсулируют техническую архитектуру внутри области, изменяя традиционную взаимосвязь.

ОБНАРУЖЕНИЕ НОВЫХ РЕСУРСОВ ПУТЕМ АВТОМАТИЗАЦИИ DEVOPS

Однажды Нил консультировал компанию, которая предлагала услуги внешнего размещения. В компании работала дюжина команд разработки, все с хорошо определенными модулями. У них была оперативная группа, которая управляла всем обслуживанием, инициализацией, мониторингом и другими задачами общего назначения. Менеджер обычно получал жалобы от разработчиков, которые хотели бы ускорить оборот необходимых ресурсов, таких как база данных и веб-серверы. Чтобы хоть как-то снизить давление, он стал назначать оперативного сотрудника один день в неделю для каждого проекта. В этот день разработчики были счастливы, поскольку могли не дожидаться ресурсов! Увы, у менеджера не было достаточно ресурсов, чтобы делать это регулярно.

Или он так считал. Мы обнаружили, что большая часть ручной работы, выполняемой операциями, была сложной случайно: неправильно сконфигурированные машины, хаос из производителей и брендов и много других поправимых нарушений. После того как все было надлежащим образом каталогизировано, мы помогли этой компании автоматизировать инициализацию новых машин, используя ситему Puppet (<https://puppet.com/>). После этой работы оперативная группа имела достаточно сотрудников, чтобы постоянно внедрять инженера по эксплуатации в каждый проект и все еще иметь достаточно людей для управления автоматизированной инфраструктурой.

Им не пришлось нанимать новых инженеров, а также менять роли. Вместо этого они воспользовались современной практикой проектирования для автоматизации всего того, чем не должен заниматься человек, освободив сотрудников для лучшего применения усилий в разработке.

Организованные бизнес-возможности

Организация команд в явном виде вокруг предметных областей означает их организацию вокруг бизнес-возможностей. Многие

организации надеются на то, что их техническая архитектура в состоянии представлять собственные сложные абстракции, слабо связанные с характеристиками предметной области, потому что разработчики традиционно подчеркивали, что чисто техническая архитектура обычно разделена функционально. Например, слоистая архитектура предназначена для упрощения обмена слоями технической архитектуры, а не для упрощения работы над объектом области, таким как Customer. Большая часть этого акцентирования появилась под воздействием внешних факторов. Например, многие типы архитектуры прошлого десятилетия по причине высоких затрат делали акцент на максимально возможном совместном использовании ресурсов.

Архитекторы постепенно избавили себя от коммерческих ограничений с помощью включения открытого доступа ко всем объектам большинства организаций. Архитектура с общими ресурсами имеет свойственные только ей проблемы, связанные с интерфейсом между частями. Теперь, когда разработчики имеют возможность создать специализированные среды и функции, им стало легче перенести акцент с технической архитектуры и в большей степени сконцентрировать внимание на предметно-центрированной архитектуре для лучшего согласования общего блока изменения в большинстве проектов программного обеспечения.



Организируйте команды вокруг бизнес-возможностей, а не функциональных обязанностей.

Продукт важнее, чем проект

Одним из механизмов, используемых многими компаниями для сдвига своих команд, является моделирование их работы в направлении *продуктов*, а не *проектов*. Проекты программного обеспечения во многих организациях имеют общий рабочий поток. Задача поставлена, команда по разработке сформирована, они работают над решением

задачи до «завершения» и всякий раз передают программное обеспечение для последующего обслуживания на весь оставшийся срок службы. Затем команда проекта приступает к решению следующей задачи.

Это вызывает множество общих проблем. Во-первых, из-за того, что команда перешла к другим проблемам, отладкой и другими работам по обслуживанию часто сложно управлять. Во-вторых, поскольку разработчики изолированы от эксплуатационных аспектов своего кода, они меньше заботятся о таких вещах, как качество. В общем, чем больше слоев между разработчиком и его исполняемым кодом, тем меньше у него связи с этим кодом. Иногда приводит к подходу «они против нас» между эксплуатационными «анклавами», что неудивительно, так как многие организации побуждают сотрудников существовать в конфликте.

Если подумать о программном обеспечении как о *продукте*, это изменит точку зрения компании в трех направлениях. Первое: продукты живут вечно, в отличие от времени выполнения проектов. Кросс-функциональные команды (часто основанные на обратном законе Конвея) ассоциируются с их продукцией. Второе: у каждого продукта есть владелец, который оказывает поддержку его использования внутри экосистемы и управляет устанавливаемыми для него требованиями. Третье: поскольку команда кросс-функциональная, она исполняет все роли, необходимые продукту (роль бизнес-аналитика, разработчика, специалиста по контролю качества, администратора баз данных, специалиста по контролю операций и т. д.).

Настоящая цель смещения с *проекта на продукт* связана с долговременным обязательством компании перед продавцом. Команды продукта берут на себя ответственность за долговременное качество своих продуктов. В свою очередь, разработчики берут на себя ответственность за качество показателей и уделяют больше внимания дефектам. Эта точка зрения также помогает команде сформировать представление о долгосрочной перспективе.

КОМАНДЫ AMAZON «НА ДВЕ ПИЦЦЫ»

Компания Amazon стала знаменита благодаря своему подходу к командам продукта, который они назвали *команды на две пиццы*. Их подход состоит в том, что ни одна команда не должна быть больше команды, которой можно скормить две крупные пиццы. Мотивация такого разделения в большей степени обусловлена общением, а не размером команды, потому что чем больше команда, тем с большим числом людей должен общаться каждый участник. Каждая команда является кросс-функциональной, и они также придерживаются принципа «ты это построил, ты на этом работаешь», то есть каждая команда полностью владеет сервисом, включая его практическую реализацию.

Имея небольшие кросс-функциональные команды, можно использовать преимущества человеческой природы. Компания Amazon с ее подходом «команда на две пиццы» скопировала поведение небольших групп приматов. В большинстве спортивных команд около 10 игроков. Антропологи убеждены, что довербальные отряды охотников были приблизительно такой же численности. Создание команд с высокой степенью ответственности усиливает прирожденное социальное поведение, делая участников более ответственными. Предположим, что разработчик в традиционной структуре проекта написал два года назад какой-то код, который аварийно завершился посреди ночи, чем вынудил кого-то из эксплуатационного отдела среагировать на поступивший ночью сигнал и устранить аварию. На следующее утро наш невнимательный разработчик может даже не осознать, что глубокой ночью случайно вызвал панику. В кросс-функциональной команде, если разработчик написал код, который вдруг был аварийно завершен, и кто-то из его команды должен был среагировать на это, то на следующее утро наш злополучный разработчик должен был бы посмотреть на грустные, уставшие глаза участников команды, которых это случайно коснулось. Это должно было вызвать у нашего рассеянного разработчика желание стать лучше.

Создание кросс-функциональных команд предотвращает поиск виноватых среди подразделений и создает в команде чувство причастности, поощряя участников работать лучше.

Работа с внешним изменением

Мы сторонники создания компонентов с низким уровнем связанности с точки зрения технической архитектуры, структуры команды и т. п. Это позволяет обеспечить максимальные возможности эволюционировать в реальном мире. При этом компоненты должны взаимодействовать между собой для совместного использования информации и решения проблем в своей области. Как мы могли бы создать компоненты, которые способны свободно эволюционировать, при этом убедиться в том, что мы сможем сохранять целостность наших точек интеграции?

Для любой области нашей архитектуры, нуждающейся в защите от побочных эффектов эволюционирования, мы создаем функции пригодности. Обычная практика в архитектурах микросервисов состоит в применении контрактов, ориентированных на потребителя (<https://martinfowler.com/articles/consumerDrivenContracts.html>), которые представляют собой атомарные функции пригодности архитектуры интеграции приложений. Рассмотрим приведенную на рис. 8.1 иллюстрацию.

На рис. 8.1 команда *поставщика* предоставляет информацию (обычно данные в упрощенном формате) каждому потребителю, *C1* и *C2*. В ориентированных на потребителя контрактах потребители информации собирают набор тестов, содержащих то, что необходимо от поставщика, и передают их поставщику, который обещает постоянно проводить их. Поскольку эти тесты содержат информацию, необходимую потребителю, поставщик может постепенно развивать свою систему так, чтобы не испортить эти функции пригодности. В сценарии, представленном на рис. 8.1, *поставщик* выполняет тесты от лица всех трех потребителей в дополнение к собственному набору тестов. Применение функций пригодности, таких как эти, неофициально называют *сетью инженерно-технической безопасности* (engineering safety net). Ведение протокола согласования интеграции должно вы-

полняться вручную, когда легче создать функции пригодности, чтобы управлять этой рутинной работой.

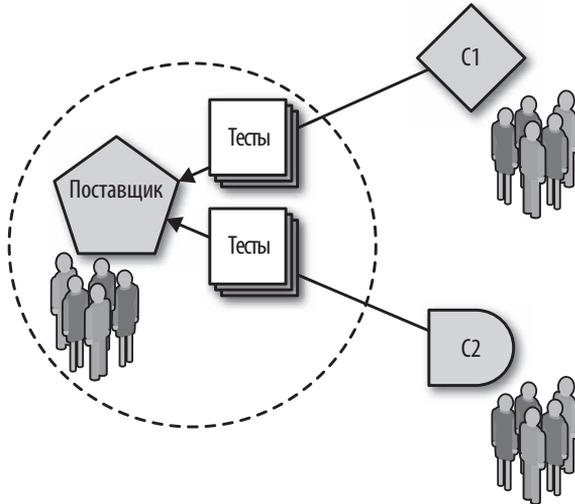


Рис. 8.1. Контракты, ориентированные на потребителя, используют тесты для заключения контрактов между поставщиком и потребителем

Одно неявное предположение, которое включено в аспект инкрементного изменения эволюционирующей архитектуры, — это определенный уровень инженерной зрелости среди разработчиков. Например, если команда использует ориентированные на потребителя контракты, но при этом могут попадаться испорченные сборки за несколько дней подряд, то команды не могут быть уверены, что их точки интеграции еще действуют. Использование процедур проектирования с охранными процедурами с помощью функций пригодности выявляет массу просчетов от разработчиков, но требует определенного уровня зрелости для успешного выполнения.

Связи между участниками команды

Многие компании случайно обнаружили, что крупные команды разработчиков не могут как следует работать, и Дж. Ричард Хакман (J. Richard Hackman), знаменитый эксперт по динамике команд,

предложил объяснение, почему это происходит. Оказывается, что эффективность работы команды зависит не от числа людей, а от количества связей, которые они должны поддерживать. Для определения того, сколько связей существует между людьми, он использовал уравнение 8.1.

Уравнение 8.1. Число связей между людьми

$$\frac{n(n-1)}{2}.$$

В уравнении 8.1 по мере роста числа людей быстро растет число связей, как показано на рис. 8.2.

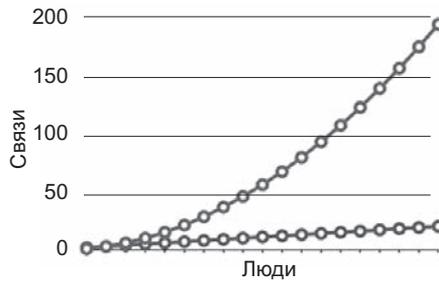


Рис. 8.2. По мере роста числа людей быстро растет число связей

На рис. 8.2 видно, что когда число членов команды достигает 20, они должны управлять 190 связями; когда же число членов команды достигает 50 человек, число связей достигает 1225. Таким образом, мотивация к созданию небольших команд связана с желанием сократить линии связи. И эти небольшие команды должны быть кросс-функциональными для устранения искусственного трения в результате координации работы отдельных подразделений.

Каждая команда не должна знать, что делают остальные команды, если только между ними не существуют точки интеграции. Но даже тогда необходимо использовать функции пригодности для гарантии целостности точек интеграции.



Стремитесь к небольшому числу связей между командами разработчиков.

Характеристики связей между командами

То, как фирмы организуют собственные структуры и управляют ими, значительно влияет на то, как создается программное обеспечение и разрабатывается архитектура. В этом разделе мы рассмотрим различные организационные и командные аспекты, которые облегчают или затрудняют разработку эволюционирующей архитектуры. Большинство разработчиков архитектур не задумываются о том, как структура команды влияет на характеристики связанности архитектуры, однако она на эти характеристики оказывает огромное влияние.

Культура

Культура (сущ.): идеи, привычки и социальное поведение определенных людей или общества.

— *Оксфордский словарь*

Архитекторы должны обращать внимание на то, как инженеры строят свою систему и следят за поведением организации. Действия и процессы принятия решений разработчиков архитектуры, используемые для выбора инструмента и создания проектов, могут оказывать большое воздействие на то, насколько хорошо программное обеспечение выносит эволюционирование. Эффективные разработчики архитектуры берут на себя ведущие роли, создавая техническую культуру и проектируя для разработчиков программного обеспечения методы создания систем. Они учат и поощряют инженеров приобретать навыки, необходимые для построения эволюционной архитектуры.

Архитектор может попытаться понять культуру проектирования команды, задавая ее участникам такие вопросы:

- Все ли в команде знают, что такое функции пригодности, и учитывают ли они влияние нового инструмента или продукта на способность развивать новые функции пригодности?
- Определяют ли команды, насколько хорошо их система соответствует определенным функциям пригодности?
- Понимают ли инженеры, что такое сцепление и связанность?
- Обсуждают ли они, подходят ли друг другу концепции области и технические принципы?
- Принимали ли команды решения, которые были основаны не на том, какую технологию они хотят изучить, а на том, есть ли возможности вносить изменения?
- Как команды реагируют на изменения в бизнес-среде? Борются ли они за внесение небольших изменений или они тратят слишком много времени на небольшие бизнес-изменения?

Регулирование поведения в команде часто включает в себя регулирование процесса работы в команде, поскольку люди реагируют на то, что их попросили сделать.

Расскажи, как ты оцениваешь меня, и я скажу, как я буду себя вести.

— Д-р Элияху Голдратт (*Eliyahu M. Goldratt*)
(синдром стога сена)

Если команда не имеет привычки вносить изменения, архитектор может ввести принципы, которые сделают это приоритетом. Например, когда команда рассматривает новую библиотеку или фреймворк, разработчик архитектуры может попросить команду явным образом оценить с помощью короткого эксперимента, как много дополнительной связанности добавит новая библиотека или фреймворк. Смогут ли инженеры без труда написать и протестировать код вне данной библиотеки или фреймворка, а также потребует ли новая библиотека

или фреймворк дополнительного времени выполнения, которое может привести к замедлению цикла разработки?

Дополнительно к выбору новых библиотек или фреймворков обзор кода позволит рассмотреть, насколько хорошо только что измененный код поддерживает новые изменения. Есть ли в системе какое-то другое место, в котором может неожиданно использоваться внешняя точка интеграции, и изменится ли затем эта точка интеграции, сколько при этом мест придется обновить? Конечно же, разработчики должны наблюдать за чрезмерным техническим усложнением системы, преждевременным добавлением дополнительной сложности или абстракций для изменения. Книга *Рефакторинг*¹ содержит такое пояснение:



ТРИ ПОПЫТКИ – И ВЫ УЖЕ ВЫПОЛНЯЕТЕ РЕФАКТОРИНГ

Первый раз, когда вы что-то делаете, вы просто это делаете. Во второй раз вы делаете что-то похожее и понимаете, что это дублирование, но все равно делаете это. В третий раз, когда вы делаете что-то аналогичное, вы уже выполняете рефакторинг.

Многие команды управляются и получают вознаграждение чаще всего за предоставление новой функциональности, при этом качество кода и аспекты эволюционирования учитываются, только если команды сделали их приоритетными. Разработчик, занимающийся эволюционирующей архитектурой, должен следить за действиями команды, которая отдает приоритет проектным решениям, помогающим с организацией эволюционирования, или находит способы поощрить это.

Культура эксперимента

Некоторые компании не в состоянии экспериментировать с требованиями к успешному эволюционированию, потому что они слишком

¹ <https://refactoring.com/>

заняты выполнением плановых работ. Успешное экспериментирование связано с регулярным выполнением небольших операций с целью опробования новых идей (с технической точки зрения и с точки зрения продукта) и для интеграции удачных экспериментов в существующие системы.

Реальная мера успеха — это число экспериментов, которые можно провести в течение 24 часов.

— *Томас Алва Эдисон (Thomas Alva Edison)*

Организации могут поддерживать экспериментирование различными путями:

Заимствование чужих идей

Многие компании направляют своих работников на конференции и поощряют их выискивать новые технологии, инструменты и методы, которые могли бы лучше решить ту или иную проблему. Другие компании используют в качестве источника новых идей рекомендации сторонних организаций или консультантов.

Поощрение явных улучшений

Компания «Тойота» знаменита своей культурой кайдзен, или непрерывным улучшением. От каждого сотрудника ожидается непрерывный поиск постоянных улучшений, в особенности таких, которые ближе всего подходят к проблемам и позволяют их решить.

Пробные и стабилизирующие решения

Пробное решение (spike solution) является практикой экстремального программирования, в которой команда получает экспериментальное решение, чтобы быстро выявлять трудно преодолимые технические проблемы, изучать незнакомую область или повышать достоверность оценок. Пробное решение повышает скорость изучения за счет качества программного обес-

печения; никому не придет в голову помещать пробное решение прямо в производство, потому что будет не хватать времени на обдумывание и на то, чтобы сделать его работоспособным. Оно создается для изучения, а не как хорошее инженерное решение.

Время для создания инновации

Компания Google хорошо известна своим правилом «20 %», когда сотрудники могут работать над сторонними проектами в течение 20 % своего времени. Другие компании организуют хакатоны¹ и дают возможность командам искать новые продукты и способы усовершенствования существующих продуктов. Компания Atlassian проводит ежеквартальные внутренние конкурсы инноваций ShipIt (<https://www.atlassian.com/company/shipit>).

Следование комплексной разработке

Комплексная разработка фокусируется на изучении нескольких подходов. На первый взгляд многочисленные варианты кажутся затратными из-за дополнительной работы, но при изучении нескольких вариантов одновременно команды завершают работу с лучшим пониманием проблемы и находят реальные ограничения на использование инструментов или методов. Ключом к эффективной комплексной разработке является построение за короткий период времени нескольких прототипов разными методами (то есть меньше чем за несколько дней), чтобы собрать более конкретные данные и получить необходимый опыт. Более надежное решение часто появляется в результате учета нескольких конкурирующих между собой решений.

Связь инженеров с конечными пользователями

Экспериментирование успешно только тогда, когда команды понимают влияние своей работы. Во многих фирмах обладающие экспериментальным мышлением сотрудники, команды и персонал, занимающийся продуктом, непосредственно видят влияние решений на конечного пользователя. Это

¹ <https://ru.wikipedia.org/wiki/Хакатон>

вдохновляет их экспериментально изучать такое влияние. А/В-тестирование¹ является одной из практик компаний с экспериментальным мышлением. Другие компании направляют команды и инженеров, чтобы наблюдать за тем, как пользователи взаимодействуют с программным обеспечением для достижения определенной задачи. Эта практика, взятая со страниц сообщества юзабилити, позволяет разработчикам поставить себя на место конечных пользователей. Инженеры возвращаются с лучшим пониманием потребностей пользователей и с новыми идеями для внедрения.

Операционный денежный поток (ОСФ) и бюджетирование

Многие традиционные функции архитектуры предприятия, такие как планирование бюджета, должны отражать изменения приоритетов в эволюционирующей архитектуре. В прошлом бюджетирование было основано на способности прогнозирования долгосрочных тенденций в экосистемах разработки программного обеспечения. Однако, как мы уже предлагали в этой книге, фундаментальная природа динамического равновесия разрушает возможность прогнозировать.

Интересная взаимосвязь существует между архитектурными квантами и затратами на архитектуру. По мере роста числа квантов затраты на квант снижаются, пока разработчики не достигают оптимальной точки, как это показано на рис. 8.3.

На рис. 8.3 видно, что по мере увеличения числа квантов архитектуры затраты на каждый квант уменьшаются по нескольким причинам. Первое: из-за того, что архитектура состоит из небольших частей, разделение ответственности должно быть более дискретным и определенным. Второе: увеличение числа физических квантов требует автоматизации работы с ними, потому что за пределами определенной

¹ <https://ru.wikipedia.org/wiki/А/В-тестирование>

области управлять рутинными операциями вручную оказывается нецелесообразно.

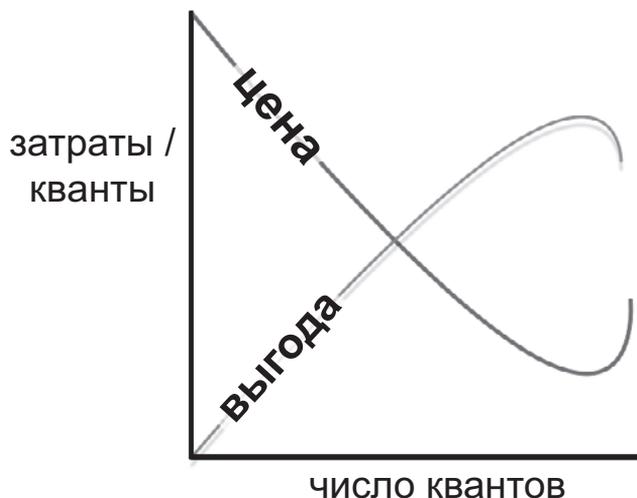


Рис. 8.3. Взаимосвязь между архитектурными квантами и затратами на архитектуру

Однако можно сделать кванты такими маленькими, что их число станет огромным, а это принесет большие затраты. Например, в архитектуре микросервисов можно построить сервисы гранулярностью в одно поле формы. На этом уровне затраты на координацию между всеми небольшими частями начинают преобладать над остальными факторами архитектуры. Поэтому в экстремумах на графике число квантов снижает выгоды в расчете на квант.

В эволюционирующей архитектуре разработчики стремятся найти оптимальную точку между надлежащим размером кванта и соответствующими затратами. Все компании отличаются друг от друга. Например, компания, работающая на активном рынке, должна будет перемещаться быстрее, и поэтому ей требуется квант меньшего размера. Следует помнить, что скорость появления нового поколения пропорциональна времени цикла, а кванты меньшего размера имеют более короткий цикл. Другой компании может оказаться целесообразно построить сервис-ориентированную архитектуру (см. главу 4)

с квантом приложения большего размера, потому что это позволяет точнее моделировать общее изменение.

Как только мы столкнулись с экосистемой, которая определяет планирование, оказалось, что наилучшее согласование между архитектурой и затратами определяют очень много факторов. Это отражает наши наблюдения, что роль разработчика архитектуры стала шире: выбор архитектуры стал влиять больше, чем когда бы то ни было.

Вместо того чтобы придерживаться практики десятилетней давности разработки архитектуры предприятий, современные архитекторы должны понимать преимущества эволюционирующих систем наряду со свойственной им неопределенностью.

Разработка функций пригодности для предприятия

В эволюционирующей архитектуре роль архитектуры предприятия вращается вокруг *управления* и *функций пригодности для всего предприятия*. Архитектура микросервисов отображает эту модель изменения. Поскольку все сервисы по операциям не связаны друг с другом, совместно используемые ресурсы не принимаются во внимание. Вместо этого разработчики обеспечивают в архитектуре управление вокруг целевых точек связанности (как в случае шаблонов сервисов) и выбор платформы. Архитектура предприятия обычно обладает собственными совместно используемыми функциями инфраструктуры и ограничивает выбор платформы тем вариантом, который поддерживается в пределах всего предприятия.

ПРАКТИЧЕСКИЙ ПРИМЕР: ДОПУСТИМОСТЬ ИСПОЛЬЗОВАНИЯ БИБЛИОТЕК С ОТКРЫТЫМ ИСХОДНЫМ КОДОМ

В какой-то момент времени юристы PenultimateWidgets задались вопросом о допустимости использования библиотек с открытым исходным кодом компанией. Они тщательно изучили лицензии на каждый фреймворк и библиотеку и установили, что компания

PenultimateWidgets не пользовалась ничем, что могло бы вызвать проблемы. Но затем один из юристов спросил: «Как мы узнаем, если у лицензии изменились условия?» Однако такого сервиса не оказалось.

Однако сразу после того, как юридический отдел сертифицировал текущие библиотеки, разработчики расположили текст лицензии внутри библиотеки и создали *временную функцию пригодности*, которая всегда проверяла бы изменения в этой строке. Таким образом, всякий раз, когда лицензия библиотеки меняется (по любой причине), функция пригодности запускает сообщение, что что-то изменилось. Конечно, функция пригодности не должна быть настолько сложной, чтобы определить, было ли изменение надлежащим, — кто-то «застрянет» на этой рутинной работе, но разработчики архитектуры могут построить функцию пригодности, которая, в отличие от автоматизированного решения, целенаправленно привлекала бы внимание.

Другую новую роль, созданную эволюционирующей архитектурой, стали играть архитектуры предприятий, которые определяют функции пригодности в пределах всего предприятия. Разработчики архитектуры предприятия обычно несут ответственность за нефункциональные требования, такие как масштабируемость и безопасность. Многие организации не имеют возможности автоматически оценивать, насколько хорошо проекты по этим характеристикам выполняются по отдельности и в совокупности. После того как проекты адаптировали функции пригодности по защите частей архитектуры, архитекторы могут применять такой же механизм для проверки сохранения целостности характеристик всего предприятия.

Если каждый проект использует конвейер развертывания для применения функций пригодности как части сборки, архитекторы могут также вставить несколько своих собственных функций пригодности. Это позволит в каждом проекте непрерывно проверять сквозную функциональность, такую как масштабируемость, безопасность и прочие функции для всего предприятия, чтобы как можно раньше обнаружить дефекты. Так же, как проекты в микросервисах используют

шаблоны услуг для унификации частей технической архитектуры, архитекторы предприятия могут использовать конвейеры развертывания для последовательного тестирования проектов.

Практический пример: PenultimateWidgets как платформа

Дела в PenultimateWidgets шли так хорошо, что компания решила продать часть своей платформы другим организациям, которые продавали аналогичные виджеты. Частью привлекательности платформы PenultimateWidgets является ее проверенная масштабируемость, отказоустойчивость, производительность и т. п. Однако разработчики архитектуры не хотели продавать эту платформу, так как не хотели потом слушать истории о возникающих ошибках в связи с тем, что пользователи начали расширять платформу разрушительным образом.

Чтобы сохранить важные характеристики платформы, разработчики PenultimateWidgets предоставили конвейер развертывания вместе с платформой и встроенными функциями пригодности для защиты важных областей. Для сохранения сертификатов пользователи платформы должны были сохранить существующие функции пригодности и (на это надеялись разработчики) добавить свои собственные, поскольку они расширили платформу.

С чего мы начнем?

Многие разработчики с существующими архитектурами, напоминающими архитектуру большого комка грязи, боролись за то, откуда начинать добавление эволюционирования. В то время как надлежащая связанность и применение модульности являются первыми шагами, которые необходимо сделать, иногда существуют другие приоритеты. Например, если схема используемых данных обладает безнадежной связанностью, определение того, как администратор базы данных может обеспечить модульность, может быть самым первым шагом. В этом разделе приводятся некоторые распространенные стратегии

и причины адаптировать практику создания эволюционирующих архитектур.

Низко висящие фрукты

Если организации необходимо скорейшее подтверждение работоспособности метода, разработчики могут выбрать самую легкую проблему, которая подчеркивает подход эволюционной архитектуры. В общем случае, это будет часть системы, которая уже не имеет значительной связанности и, вероятно, не представляет собой критичный путь к любой зависимости. Повышение модульности и снижение связанности дает возможность командам продемонстрировать другие аспекты эволюционирования архитектуры, а именно функции пригодности и инкрементные изменения. Разработка улучшенной изоляции позволяет больше сосредоточиться на тестировании и создании функций пригодности. Улучшенная изоляция развертываемых блоков облегчает создание конвейера развертывания и обеспечивает платформу для более надежного тестирования.

Метрики являются общим дополнением к конвейеру развертывания в средах с инкрементным изменением. Если команды используют эти усилия в качестве доказательства концепции, разработчики должны будут собрать соответствующие метрики для использования сценариев перед и сценариев после. Сбор конкретных данных является для разработчиков лучшим способом проверки достоверности метода; помните, что *демонстрация побеждает споры*.

Такой подход, «сначала самое простое», минимизирует риск при возможных издержках стоимости, если, конечно, команде удастся сбалансировать *простоту и высокую ценность*. Это хорошая стратегия для компаний, которые настроены скептически и хотят окунуться в аллегорическую воду эволюционной архитектуры.

Максимальная ценность

Альтернативой подходу «сначала самое простое» является подход «сначала самое ценное», в котором предполагается найти наиболее

критическую часть системы и сначала создать в ней эволюционирование характеристик. Компании могут выбрать этот подход по нескольким причинам. Первое: если разработчики архитектуры убеждены, что они хотят добиться эволюционирования архитектуры, то выбор в начале максимальной ценности указывает на их ответственность. Во-вторых, для архитекторов компаний, которые все еще оценивают эти идеи, может быть интересно, насколько применимы эти методы в пределах их экосистемы. Поэтому, выбирая в начале максимальную ценность, они демонстрируют свою уверенность в долговременной ценности эволюционирующей архитектуры. Третье: если у разработчиков архитектуры есть сомнения в том, что эти идеи могут работать для их приложений, проверка этих концепций с помощью наиболее ценной части своей системы дает данные, позволяющие принять решение, хотят ли они продолжить и дальше их применение.

Тестирование

Многие компании жалуются на отсутствие тестирования своих систем. Если разработчики программного обеспечения обнаружили слабую базу кода или отсутствие тестирования, то они могут принять решение добавить некоторые решающие тесты, прежде чем предпринимать более существенные шаги в направлении к реализации эволюционирующей архитектуры.

Разработчики обычно не одобряют решения приниматься за проект, в котором тесты были добавлены только в базу кода. Руководство смотрит на эту активность с подозрением, особенно если это приводит к задержке введения новых функций. Наоборот, разработчики архитектуры должны сочетать повышение модульности с высокоуровневым функциональным тестированием. Комбинирование функционального тестирования с модульным обеспечивает лучшую кодогенерацию для процедур проектирования, таких как разработка на основе тестирования (TDD), однако отнимает время для встраивания в базу кода. Вместо этого разработчики должны добавить крупномодульное функциональное тестирование для некоторых

характеристик, реконструируя этот код, предоставляя возможность проверить, что поведение всей системы в целом не изменилось в результате реструктуризации.

Тестирование является важным компонентом процесса инкрементного изменения эволюционирующей архитектуры, а функции пригодности активно воздействуют на тестирование. Поэтому какой-то уровень тестирования позволяет использовать эти методы, а между полнотой тестирования и тем, насколько легко удастся реализовать эволюционирующую архитектуру, существует корреляция.

Инфраструктура

В некоторые компании новые возможности приходят медленно, и оперативная группа становится жертвой недостатка инноваций. Для компаний, не имеющих функциональной инфраструктуры, решение этих проблем может стать начальным этапом создания эволюционирующей архитектуры. Проблемы с инфраструктурой проявляются в разных формах. Например, некоторые компании передают все свои оперативные работы другой компании и, таким образом, не контролируют эту критическую часть своей экосистемы; сложность DevOps возрастает на порядок при обременении накладных расходов на координацию компаний.

Другой распространенной нефункциональной инфраструктурой является защищенный брандмауэр между разработкой и операциями, когда разработчики не понимают, как фактически выполняется код. Эта структура является распространенной в компаниях, в которых каждое подразделение действует автономно.

И наконец, архитекторы и разработчики в некоторых организациях игнорируют общепринятые практики, и в результате этого создают большой технический долг, который проявляется в инфраструктуре. В некоторых компаниях даже нет представления о том, что и где работает, у них отсутствуют и другие базовые знания взаимодействий между архитектурой и инфраструктурой.

ИНФРАСТРУКТУРА МОЖЕТ ВЛИЯТЬ НА АРХИТЕКТУРУ

Однажды Нил консультировал компанию, которая предлагала услуги внешнего размещения. В этой компании использовалось большое число серверов (на тот момент около 2500) и имелись подразделения *внутри* оперативных групп: одна команда устанавливала аппаратные средства, другая — операционные системы, а третья занималась установкой приложений. Конечно же, когда разработчику нужны ресурсы, он бросает билет в черную дыру операций, в которой появляется много таких билетов, неделями болтающихся там, пока не появляются ресурсы. Проблема усугубилась тем, что IT-директор компании покинул ее годом раньше, а с IT-отделом работал финансовый директор. Конечно же, финансовый директор был озабочен в основном сокращением расходов, а не модернизацией того, что он рассматривал просто как накладные расходы.

При исследовании слабых мест в работе один из разработчиков отметил, что на каждом сервере размещено около пяти пользователей, что было шокирующим, учитывая простоту приложения. Разработчики смущенно объяснили, что они неправильно использовали состояние сеанса в HTTP-протоколе, в основном рассматривая сеансы как крупную базу данных в памяти. Поэтому они могли принимать всего нескольких пользователей в расчете на сервер. Проблема заключалась в том, что их оперативная группа не смогла создать реалистичную производственную среду для целей отладки, и они полностью отстранили разработчиков от процесса отладки (или даже от всестороннего мониторинга), в основном по конъюнктурным соображениям. Не имея возможности взаимодействовать с реалистичной версией приложения, разработчики не могли распутать все то, что они постепенно создавали.

Выполняя оценку вычислений, мы установили, что компания, вероятно, могла бы работать на серверах, число которых было бы на порядок меньше, например 250. Однако компания была слишком занята покупкой новых серверов, установкой операционных систем и т. п. По иронии, предпринятые компанией меры по сокращению расходов фактически обошлись ей в огромную сумму.

В конечном счете осажденные разработчики создали свою собственную партизанскую группу DevOps и начали управлять серверами сами, обходя все традиционные операции организации. Между двумя этими группами в будущем началась битва, но вскоре разработчики добились улучшения в реструктурировании своего приложения.

В конце концов, раздражающий, но точный ответ приглашенного консультанта был *«всякое бывает!»*. Только разработчики архитектуры, разработчики программного обеспечения, ДВА, члены групп DevOps, тестирований, безопасности и все остальные участники процесса разработки и эксплуатации могут окончательно определить наилучшую дорожную карту в направлении к эволюционирующей архитектуре.

Практический пример: архитектура предприятия в компании PenultimateWidgets

В PenultimateWidgets считают, что необходимо реконструировать значительную часть их прежней платформы, и команда архитекторов предприятия создала электронную таблицу с перечнем всех свойств, которые должны быть у новой платформы: безопасность, метрики производительности, масштабируемость, способность к развертыванию и масса других свойств. Каждая категория содержит до 20 ячеек, все они — с определенными критериями. Например, один из показателей времени безотказной работы требует, чтобы каждый сервис давал пять девяток (99,999) пригодности. Всего ими было определено 62 отдельных пункта.

Но они осознали некоторые проблемы с этим подходом. Первое, будут ли проверяться все 62 свойства в проекте? Они могли бы создать соответствующие правила, но кто будет оперативно проверять, что эти правила выполняются? Проверка всех приведенных выше характеристик, даже на специальной основе, была бы значительной проблемой.

Второе, имело ли бы смысл накладывать строгие указания по пригодности для каждой части системы? Действительно ли важно, чтобы администратор отсеивал предложения по пяти девяткам? Политика тотального контроля часто ведет к очевидному чрезмерному техническому усложнению системы.

Для решения этих проблем разработчики архитектуры предприятия задают критерии, такие как функции пригодности, и создают шаблон конвейера развертывания, с которого начинается каждый проект.

Внутри конвейера развертывания разработчики архитектуры проектируют функции пригодности для автоматической проверки важных функций, таких как безопасность, оставляя отдельным командам возможность добавлять специальные функции пригодности (такие, как пригодность) для своих сервисов.

Будущее состояние?

Каким будет будущее состояние эволюционирующей архитектуры? По мере того как команды ближе знакомятся с идеями и практиками, они будут как обычно включать их в свою предметную область и начинать применять эти идеи для получения новых возможностей, таких как развитие на основе данных.

Большую работу необходимо сделать для более сложных функций пригодности, но определенный прогресс появляется по мере того, как организации решают проблемы и открытый доступ является одним из многих решений. В первые годы эпохи гибкости люди жаловались, что некоторые задачи было трудно автоматизировать, но отважные разработчики продолжали расчищать себе дорогу, и теперь все центры обработки и хранения данных поддаются автоматизации. Например, компания Netflix внесла грандиозную инновацию в концептуализацию и создание таких инструментов, как Simian Army, которые поддерживают комплексную непрерывную функцию пригодности (которая еще так не называлась).

Существует две многообещающие области.

Функции пригодности, использующие искусственный интеллект

Постепенно фреймворки с открытым исходным кодом, в которых используется искусственный интеллект (ИИ), становятся доступны для регулярных проектов. По мере того как разработчики знакомятся с использованием этих инструментов разработки программного обеспечения, мы предусмотрим функции пригодности на основе ИИ,

которые следили бы за аномальным поведением. Компании, работающие с кредитными картами, уже применили эвристический подход, такой как маркировка одновременных транзакций в различных частях мира; разработчики архитектур могут начать создавать инструменты расследования для отслеживания странного поведения в архитектуре.

Генеративное тестирование

Практика, которая была принята во многих сообществах функционального программирования и получила широкое признание, основана на идее *генеративного тестирования* (generative testing). Традиционные модульные тесты включают в себя подтверждение правильности выходных данных в каждом тесте. Однако, используя генеративное тестирование, разработчики выполняют большое число тестов и собирают выходные данные, а затем применяют статистический анализ полученных результатов, чтобы выявить аномалии. Например, рассмотрим обычный случай граничной проверки диапазонов чисел. Традиционные модульные тесты проверяют известные места, в которых прерываются числа (появляются отрицательные величины, превышаются размеры и т. п.), однако они невосприимчивы к непредвиденным пограничным случаям. Генеративное тестирование проверяет каждое возможное значение и сообщает о пограничных случаях, которые были нарушены.

Зачем это (или почему бы и нет)?

Не существует «серебряной пули», и архитектура не является исключением. Мы не рекомендуем в каждом проекте добиваться способности эволюционирования за счет дополнительных затрат и усилий, если это не приносит преимуществ.

Зачем та или иная компания решает строить эволюционирующую архитектуру?

Многие компании считают, что за последние несколько лет цикл изменений ускорился, что отражено в вышеупомянутой заметке *Forbes*,

что каждая компания должна быть компетентной в разработке и доставке программного обеспечения.

Прогнозируемая или развиваемая

Многие компании оценили долговременное планирование ресурсов и прочих стратегических факторов; очевидно, что компании ценят *прогнозируемость*. Однако из-за динамического равновесия экосистем разработки программного обеспечения прогнозируемость пропала. Разработчики архитектуры предприятия все еще могут строить планы, но они в любой момент могут потерять актуальность.

Даже компании из устоявшихся, развитых отраслей не станут игнорировать риски систем, которые не могут развиваться. Такси было многоцентровым международным институтом, когда его начали сотрясать компании, занимающиеся райд-шерингом, то есть совместными поездками. Эти компании поняли и отреагировали на изменяющуюся экосистему. Такое явление известно как дилемма инноватора¹. Оно демонстрирует, что компании в установившихся рынках могут прогореть, поскольку более гибкие стартапы лучше реагируют на меняющуюся экосистему.

Построение эволюционирующих архитектур требует дополнительного времени и усилий, но награда приходит, когда компания может реагировать на существенные изменения рынка без значительных переделок. Предсказуемость никогда не вернет нас в эпоху мейнфреймов и специализированных вычислительных центров. Высоковолатильная природа мира разработок все сильнее подталкивает все организации к инкрементным изменениям.

Масштаб

Долгое время наилучшей практикой в архитектуре считалось построение системы транзакций с опорой на реляционные базы данных, с использованием многих функций баз данных для управления ко-

¹ https://ru.wikipedia.org/wiki/Подрывные_инновации

ординацией. Проблемой этого подхода является масштабирование, так как с его применением становится тяжело масштабировать бэкенд базы данных. Для решения этой проблемы было создано множество хитроумных технологий, но все они лишь частично решали фундаментальную проблему масштабирования, связанности. Любая соединительная точка в архитектуре в конечном счете препятствует масштабированию, а надежда на координацию базы данных приводит в тупик.

Компания Amazon столкнулась с этой проблемой. Исходный сайт был спроектирован с монолитным фронтендом, привязанным к монолитному бэкенду, смоделированному для баз данных. При увеличении трафика они должны были увеличить масштаб баз данных. В какой-то момент они достигали предельных масштабов баз данных, а воздействие на их сайт снижало производительность — все страницы загружались медленнее.

В Amazon осознали, что связывание всего с чем-то *одним* (с реляционной базой данных, шиной сервиса предприятия и т. п.) ограничивает масштабируемость. С помощью перепроектирования текущей архитектуры в архитектуру микросервисов, которое устраняло ненадлежащую связанность, они обеспечили своей экосистеме возможность масштабирования.

Побочным преимуществом уменьшения связанности системы этого уровня является усиленное эволюционирование. На протяжении всей книги мы показывали, что ненадлежащая связанность представляет самую большую проблему эволюционирования. Построение масштабируемой системы также стремится соответствовать эволюционируемой системе.

Передовые бизнес-возможности

Многие компании смотрят с завистью на Facebook, Netflix и другие передовые технологические компании, потому что они пользуются технически сложными функциями. Инкрементные изменения позволяют использовать хорошо известные практики, такие как разработки

на основе гипотез и на основе данных. Многие компании стремятся включить в цепь обратной связи своих пользователей с помощью многомерного тестирования. Основным строительным блоком для многих современных практик DevOps является архитектура, которая может эволюционировать. Например, оказалось, что разработчикам трудно выполнять А/В-тестирование при наличии между компонентами высокой степени связанности, что делает затруднительной изоляцию зон ответственности. Обычно эволюционирующая архитектура позволяет компании лучше реагировать на неизбежные, но непредсказуемые изменения.

Время цикла как бизнес-метрика

В разделе «Конвейеры развертывания» (с. 64) было показано различие между процессом *непрерывной поставки*, в котором минимум один этап в конвейере развертывания выполняет ручную *поставку*, и процессом *непрерывного развертывания*, в котором переход с каждого этапа при его успешном завершении на следующий выполняется автоматически. Построение непрерывного развертывания требует высокого уровня выполнения проектных работ; зачем же компаниям так далеко заходить?

Эта необходимость связана с тем, что на некоторых рынках время цикла становится фактором, определяющим различия в бизнес-среде. Некоторые крупные консервативные организации смотрят на программное обеспечение как на накладные расходы и поэтому пытаются минимизировать затраты. Инновационные компании видят в программном обеспечении преимущество, обеспечивающее конкурентоспособность. Например, если компания AcmeWidgets разработала архитектуру со временем цикла в три часа, а у PenultimateWidgets он остается равным шести неделям, у AcmeWidgets есть преимущество, которым она может воспользоваться.

Многие компании рассматривают время цикла как первоклассный показатель развития бизнеса в основном потому, что они живут в условиях высокой рыночной конкуренции. Все рынки в конечном счете становятся, таким образом, конкурентными. Например,

в начале 1990-х годов некоторые крупные компании вели себя активнее на пути автоматизации выполняемого вручную рабочего потока, используя для этого программное обеспечение, и получали огромные преимущества по мере того, как все компании осознавали необходимость этого.

Изоляция характеристик архитектуры на уровне кванта

Представление традиционных нефункциональных требований в виде функции пригодности и построение хорошо инкапсулированного архитектурного кванта дает возможность разработчикам поддерживать различные характеристики каждого кванта, что является одним из преимуществ микросервисной архитектуры. Поскольку техническая архитектура каждого кванта не связана с остальными квантами, разработчики архитектуры могут выбирать для каждого варианта использования разные архитектуры. Например, разработчики для одного небольшого сервиса могут выбрать микроядерную архитектуру, так как они хотят поддерживать небольшое ядро, позволяющее постепенные добавления. Другая группа разработчиков может выбрать для своего сервиса архитектуру на основе событий, что связано с проблемами масштабирования. Если оба сервиса являются частью монолитной архитектуры, разработчикам придется идти на компромиссы, чтобы удовлетворить оба требования. Изолируя техническую архитектуру на уровне небольшого кванта, разработчики архитектуры могут сосредоточиться на основных характеристиках одного кванта, не анализируя компромиссы для приоритетов конкуренции.

Практический пример: избирательный масштаб в PenultimateWidgets

У PenultimateWidgets есть сервисы, которым достаточно совсем немного в плане масштабирования, и поэтому эти сервисы написаны в простых технологических стеках. Однако пара сервисов отличаются от них. Сервис `Import` должен каждую ночь импортировать данные инвентаризации из салонов обслуживания для системы бухгалтерского учета. Поэтому архитектурные характеристики и функции при-

годности, которые разработчики встроили в сервис **Import**, включают *масштабируемость* и *отказоустойчивость*, которые сильно усложняют техническую архитектуру этого сервиса. Другой сервис, **MarketingFeed**, вызывает каждый салон при его открытии для получения сведений о ежедневных продажах и обновлениях. Сервис **MarketingFeed** нуждается в *способностях к быстрой адаптации*, чтобы справляться с всплесками запросов по мере открытия салонов в разных часовых поясах.

Распространенной проблемой в архитектуре с высокой степенью связанности является непреднамеренное чрезмерное техническое усложнение системы. В еще более связанной архитектуре разработчикам потребуется встроить в каждый сервис такие характеристики, как масштабируемость, отказоустойчивость и гибкость, значительно усложняя те сервисы, которым это необходимо. Разработчики привыкли выбирать архитектуру исходя из набора соответствующих компромиссов. Разработка архитектуры с четко определенными границами кванта позволяет выбрать точные характеристики, которые требуются архитектуре.

Адаптация и эволюционное развитие

Многие организации попались в ловушку постепенного увеличения технического долга и нежелания выполнять необходимую реструктуризацию, что, в свою очередь, делает системы и точки интеграции хрупкими. Компании стараются обойти эту хрупкость с помощью таких инструментов, как шины сервиса, которые уменьшают головную боль, но не рассматривают проблему глубже логического сцепления бизнес-процессов. Использование шины сервиса является примером *адаптации* существующей системы для применения при других настройках. Но, как мы уже подчеркивали раньше, побочным эффектом адаптации является увеличение технического долга. Когда разработчики что-то адаптируют, они сохраняют исходные характеристики и добавляют вместе с ними слой с новыми характеристиками. Чем больше циклы адаптации выдерживает компонентов, тем более параллелизма будет в поведении архитектуры, что приведет к повышению сложности, но надеемся, что стратегически.

Использование переключателей функций дает хороший пример преимуществ адаптации. Часто разработчики используют переключатели, когда пытаются использовать несколько альтернатив с помощью разработки на основе гипотез, проверяя на пользователях, какая из альтернатив лучше. В этом случае технический долг, налагаемый переключателями, является умышленным и желательным. Конечно, наилучшие практики проектирования такого типа переключателей сводятся к тому, что их необходимо сразу же убрать, после того как с их помощью было получено решение.

Альтернативно этому, *эволюционное развитие* предусматривает фундаментальное изменение. Построение эволюционирующей архитектуры тянет за собой изменение архитектуры на месте, обеспечивая защиту от повреждений функциями пригодности. Окончательным результатом является система, которая продолжает эволюционировать, не приводя к увеличению наследуемых характеристик в обновляемых решениях.

По какой причине компания делает выбор не строить эволюционирующую архитектуру?

Мы не уверены, что эволюционирующая архитектура — лекарство от всех болезней. У компании может быть несколько причин не реализовывать эти идеи.

Нельзя эволюционировать комок грязи

Одной из ключевых возможностей, которую разработчики архитектуры игнорируют, является *возможность технической реализации* — следует ли команде браться за этот проект? Если архитектура представляет собой безнадежно связанный большой комок грязи, то превращение его в эволюционирующую систему потребует огромного объема работ, вероятно, большего, чем переписывание с нуля. Компании не склонны выбрасывать то, что еще имеет субъективную ценность, но часто переделка оказывается более затратной, чем переписывание заново.

Как компании могут понять, что они оказались в этой ситуации? Первым шагом превращения существующей архитектуры в эволюционирующую систему является *модульность*. Поэтому первой задачей разработчика является поиск существующей в каком-либо виде модульности в текущей системе и реструктурирование архитектуры на основе найденной модульности. После того как архитектура стала менее путанной, разработчику становится легче разглядеть лежащую в ее основании структуру и сделать обоснованные оценки необходимых усилий для реструктуризации.

Доминируют другие характеристики архитектуры

Способность к эволюционированию лишь одна из немногих характеристик, которые разработчик должен оценить при выборе той или иной архитектуры. Ни одна архитектура не может полностью поддерживать конфликтующие основные цели. Например, встраивание высокой производительности и масштабируемости в одну и ту же архитектуру затруднительно. В некоторых случаях могут перевесить другие факторы.

Большую часть времени разработчики выбирают архитектуру для выполнения обширного набора требований. Допустим, что архитектура должна поддерживать высокую степень пригодности, безопасности и масштабирования. Это приводит к общим паттернам архитектуры, таким как монолитная архитектура, архитектура микросервисов или архитектура на основе событий. Однако семейство архитектур, *зависящих от предметной области*, стремится максимизировать одну из характеристик.

Превосходным примером зависящей от предметной области архитектуры является архитектура LMAX¹ компании Custom Trading Solution, Inc. Основной целью компании было обеспечить пропускную способность быстрых транзакций, и они безуспешно экспериментировали с разными методами. В конце концов, проанализировав

¹ <https://martinfowler.com/articles/lmax.html>

нижний уровень, они обнаружили ключ к масштабируемости, который делал их логическую схему достаточно небольшой, чтобы она подошла для кеша ЦПУ, и предварительно распределили всю память, чтобы исключить скопление мусорных данных. Их архитектура достигла ошеломляющих 6 миллионов транзакций в секунду в одном Java-потоке!

Если архитектура построена для такой специализированной цели, то ее развитие для дальнейшего включения других функций будет затруднительно (если только разработчикам не повезет и архитектурные функциональности частично не наложатся друг на друга). Поэтому большинство зависящих от предметной области архитектур не связаны с эволюционированием, так как их цель преобладает над остальными.

Жертвенная архитектура

Мартин Фаулер дал определение жертвенной архитектуре¹, как предназначенной для того, чтобы ее выбросить. Многим компаниям требуется построить вначале простые версии для исследования рынка или проверки жизнеспособности. После успешной проверки они могут построить *настоящую* архитектуру для поддержки тех характеристик, которые проявились при исследовании.

Многие компании делают это стратегически. Часто компании строят такой тип архитектуры при разработке минимально жизнеспособного продукта для тестирования рынка, планируя в дальнейшем построить более надежную архитектуру, если рынок одобрит представленный продукт. Построение жертвенной архитектуры предполагает, что разработчики не собираются ее эволюционировать, а собираются заменить на что-то более постоянное. Облачные предложения делают это привлекательным вариантом для компаний, экспериментирующих с жизнеспособностью нового рынка или предложения.

¹ <https://martinfowler.com/bliki/SacrificialArchitecture.html>

Заккрытие бизнеса в планах

Эволюционирующая архитектура помогает бизнесу адаптироваться к изменениям экосистемы. Если компания не планирует вести дела через год, то нет причин заниматься построением эволюционной архитектуры.

Некоторые компании находятся в таком положении; просто они этого еще не осознали.

Убеждая других

Архитекторы и разработчики борются за то, чтобы нетехнические менеджеры и их коллеги понимали преимущества эволюционной архитектуры. Особенно это относится к тем частям организации, которые в наибольшей степени оторваны от необходимых изменений. Например, разработчики, которые говорят операционной группе о некорректном выполнении своей работы, встретят сопротивление.

Мы рассказали о наилучшем решении этой проблемы в главе 6. Вместо того чтобы *убеждать*, лучше *продемонстрируйте то*, как эти идеи улучшат их работу на практике.

Практический пример: консультация по системе дзюдо

Одна из наших сотрудниц работала с крупной розничной сетью, пытаясь убедить разработчиков архитектуры предприятия и оперативную группу воспользоваться более современной практикой DevOps, такой как автоматизированная инициализация машин, усиление контроля и т. п. Однако все ее предложения не находили отклика, что выражалось двумя простыми фразами: «У нас нет времени» и «Наши схемы слишком сложные, все это не будет здесь работать».

Тогда она применила превосходную технику, названную *консультация по системе дзюдо*. В многочисленных приемах дзюдо против соперника используется его собственный вес. *Консультация по системе дзюдо* сводится к нахождению болевой точки и фиксации ее в качестве

примера. Болевой точкой в розничной сети была среда контроля качества: ее никогда не было достаточно. Поэтому когда команды будут пытаться делиться своей рабочей средой, это будет вызывать только головную боль. Приведя такой пример, наша сотрудница получила одобрение на создание среды контроля качества с использованием современных инструментов и методов DevOps.

Когда работа была завершена, она продемонстрировала ошибочность обоих предыдущих предположений. Теперь любая команда, которой необходима среда контроля качества, могла без труда ее получить. Ее усилия, в свою очередь, убедили оперативную группу сделать дополнительные инвестиции в современные методы ввиду их очевидной ценности. Демонстрация победила обсуждения.

Пример из бизнеса

Представители бизнеса часто с подозрением относятся к амбициозным IT-проектам, которые похожи на дорогостоящее упражнение по переделке связующего программного обеспечения. Однако многие компании обнаружили, что большинство требуемых возможностей имеют основу в эволюционирующей архитектуре.

«Будущее уже наступило...»

Будущее уже наступило. Просто оно еще неравномерно распределено.

— Уильям Гибсон (*William Gibson*)

Многие компании рассматривают программное обеспечение как накладные расходы, например, как систему отопления. Когда разработчики программного обеспечения общаются с руководителями этих компаний об инновациях в программном обеспечении, они представляют себе сантехников, продающих им довольно дорогие накладные расходы. Однако это устаревшее представление о стратегической важности программного обеспечения дискредитировано. Лица, принима-

ющие решения относительно закупок программного обеспечения, как правило, становятся институционально консервативными, оценивая экономию затрат на инновации. Архитекторы предприятия совершают эту ошибку по понятным причинам — они смотрят на другие компании в своей экосистеме, чтобы увидеть, как они подходят к этим решениям. Но такой подход опасен, потому что пробивная компания, имеющая современную архитектуру программного обеспечения, может перейти в область существующей компании и внезапно стать доминирующей, потому что у них есть лучшие информационные технологии.

Двигаться быстро и без аварий

Большинство крупных компаний жалуются на темп изменений внутри организации. Один из побочных эффектов построения эволюционирующей архитектуры проявляется как повышение эффективности проектирования. Все практики, которые мы называем *инкрементные изменения*, улучшают автоматизацию и эффективность. Определение обязанностей архитектуры предприятия верхнего уровня как функций пригодности одновременно унифицирует различные наборы обязанностей и заставляет разработчиков думать в отношении объективной результативности.

Построение эволюционирующей архитектуры предполагает, что все команды могут с уверенностью вносить в архитектуру инкрементные изменения. В главе 2 мы описали пример GitHub, в котором использовался функциональный компонент архитектуры без регрессий (одновременно раскрывающий другие нераскрытые ошибки). Представители бизнеса боятся разрушающих изменений. Если разработчики создают архитектуру, которая позволяет вносить инкрементные изменения с большей уверенностью, чем в старой архитектуре, то от этого выигрывают и бизнес, и технология.

Меньше риска

С улучшенными практиками проектирования снижается риск. Эволюционирующая архитектура заставляет команды пользоваться современными практиками, побочный эффект которых выражается

в инкрементных изменениях. После того как разработчики стали уверены, что их практики позволят им вносить изменения в архитектуру ничего не разрушая, компании смогут увеличить частоту релизов.

Новые возможности

Лучший способ продать идею эволюционирующей архитектуры бизнесу связан с получением новых возможностей, таких как разработка на основе гипотез. У представителей бизнеса стекленеет взгляд, когда разработчики архитектуры красочно описывают технические усовершенствования, поэтому лучше сформулировать возможности от нововведений на языке бизнеса.

Построение архитектуры с эволюционным развитием

Наши идеи о построении эволюционирующей архитектуры основаны на многих существующих вещах: тестировании, метриках, конвейерах развертывания и массе других вспомогательных инфраструктур и инноваций. Мы создаем новую перспективу для унификации предыдущих разнообразных концепций, используя функции пригодности. Для нас все, что способно проверить архитектуру, является функцией пригодности, а рассмотрение всех этих механизмов облегчает автоматизацию и проверки.

Хотелось бы, чтобы разработчики архитектуры начали думать об архитектурных характеристиках как о *поддающихся оценке* параметрах, которые позволят им построить более отказоустойчивую архитектуру, а не как о необдуманном устремлении.

Создать некоторые системы более эволюционирующими будет нелегко, но у нас действительно нет выбора: экосистема разработки программного обеспечения продолжает выдавать новые идеи, которые появляются там, где их совсем не ждешь. Организации, способные реагировать и преуспевать в этой среде, будут иметь серьезные преимущества.

Об авторах

Нил Форд (Neal Ford) является директором, разработчиком архитектур программного обеспечения и идейным вдохновителем в компании ThoughtWorks. Компания занимается разработкой ПО и представляет сообщество увлеченных, нацеленных на результат людей, думающих о перспективных технологиях и способных решать сложные технические проблемы, привнося в IT-отрасль революционные идеи, которые приводят к положительным социальным изменениям. До работы в ThoughtWorks Нил был главным технологом в фирме DSW Group, Ltd., занимающейся обучением и разработками.

Нил имеет степень в области Computer Science Университета штата Джорджия, со специализацией в области языков программирования и компиляторов, а также в области статистического анализа. Он является международным признанным экспертом в разработках и поставках программного обеспечения, особенно на стыке методов гибкого проектирования и архитектур программного обеспечения. Нил является автором журнальных статей, семи книг, а также множества видеопрезентаций; был докладчиком на сотнях конференций разработчиков по всему миру. Темы этих работ включают: архитектуру программного обеспечения, непрерывную доставку, функциональное программирование, а также передовые инновации программного обеспечения. Его основная консалтинговая деятельность сосредоточена на проектировании и построении крупномасштабных приложений предприятий. Если у вас появился интерес к творчеству Нила, посетите его страничку в интернете по адресу nealford.com.

Д-р Ребекка Парсонс (Dr. Rebecca Parsons) является техническим директором ThoughtWorks с десятилетним опытом разработки приложений для различных отраслей и систем. Ее технический опыт включает в себя создание крупномасштабных приложений распределенных объектов, интеграцию разнородных систем и работу с командами по разработке архитектур. Помимо ее увлеченности наукоемкими отраслями, д-р Парсонс является активным сторонником многообразия в информационных технологиях.

До работы в ThoughtWorks д-р Парсонс работала ассистентом профессора Computer Science в Университете Центральной Флориды, где она вела курсы по компиляторам, оптимизации программ, распределенным вычислениям, языкам программирования, теории вычислений, машинному обучению и вычислительной биологии. Она также работала в качестве директора постдокторанта в национальной лаборатории Лос-Аламоса, исследуя проблемы в параллельных и распределенных вычислениях, в генетических алгоритмах, вычислительной биологии и в нелинейных динамических системах.

Д-р Парсонс получила степень бакалавра Computer Science и экономики в Университете Брэдли и степень магистра Computer Science в Университете Райса, а также степень доктора Computer Science в Университете Райса. Она является соавтором таких публикаций, как *Языки, зависящие от предметной области* (Domain-Specific Languages), *Антология ThoughtWorks* (The ThoughtWorks Anthology) и *Построение эволюционных архитектур* (Building Evolutionary Architectures).

Патрик Куа (Patrick Kua) является главным техническим консультантом компании ThoughtWorks. Он проработал в технологической отрасли свыше 15 лет. Хорошо известен своими работами по достижению баланса в сочетании техники, людей и технологического процесса для повышения эффективности поставок программного обеспечения. Часто выступает на конференциях по вопросам технического руководства, а также по архитектуре программного обеспечения и по созданию передовой культуры проектирования.

Он автор *Справочника по ретроспективе: Руководство для гибких команд и беседы с техническими руководителями: от новаторов до*

специалистов-практиков (The Retrospective Handbook: A Guide for Agile Teams and Talking with Tech Leads: From Novices to Practitioners); организовал регулярную программу обучения для поддержки разработчиков, которые становятся техническими руководителями и/или разработчиками архитектур.

Дополнительную информацию о нем можно найти на веб-сайте thequa.com, а также в «Твиттере» по адресу [@patkua](https://twitter.com/patkua) (<https://twitter.com/patkua>).

Выходные данные

На обложке книги *Эволюционная архитектура* изображен раскрывшийся коралл-мозговик (Трахифиллия Жофруа, *Trachyphyllia geoffroyi*). Он также известен как коралл «складчатый мозг», или «кратер»; это каменный крупнополипный коралл LPS, родиной которого является Индийский океан. Знаменитый своими характерными складками, яркой окраской и жизнестойкостью, этот свободноживущий коралл существует на поверхностном слое симбиотической бурой водоросли за счет фотосинтеза в течение светового дня. Ночью он вытягивает свои щупальца и направляет себе в рот (у некоторых кораллов-мозговиков два или даже три рта) различный планктон и мелкую рыбу.

Благодаря своей впечатляющей внешности и простой диете *Trachyphyllia geoffroyi* пользуется популярностью у аквариумистов. Его помещают на нижнем слое песка и/или ила, напоминающем мелководье в его естественной среде обитания. Кораллы извлекают для себя пользу из среды с умеренным течением воды, богатой планктоном и остатками живых организмов.

Trachyphyllia geoffroyi занесен в Красную книгу Международного союза охраны природы и природных ресурсов как вид, близкий к исчезновению. Многие из животных на обложках книг издательства O'Reilly находятся под угрозой; все они очень важны для нашего мира. Чтобы узнать больше о том, как можно помочь им, см. animals.oreilly.com.

Изображение на обложке взято у Jean Vincent Félix Lamouroux's *Exposition Methodique des genres de L'Ordre des Polypiers*.

Нил Форд, Ребекка Парсонс, Патрик Куа
**Эволюционная архитектура.
Поддержка непрерывных изменений**

Серия «Бестселлеры O'Reilly»

Перевел с английского *А. Демьяников*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Научный редактор	<i>П. Ковалёв</i>
Литературный редактор	<i>А. Бульченко</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>С. Беляева, И. Тимофеева</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 10.2018. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 27.09.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930.
Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: www.piter.com
- по электронной почте: books@piter.com
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

- БЕСПЛАТНАЯ ДОСТАВКА:**
- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
 - почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, гоб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гоб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гоб. 6217;
e-mail: kuznetsov@piter.com