ГЛАВА 1

Архитектура эволюционных систем

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

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

Сложности создания эволюционных систем

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

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

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

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

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

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

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

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

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

ПОЧЕМУ В 2000 ГОДУ НЕ БЫЛО МИКРОСЕРВИСОВ

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

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

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

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

«Уходите».

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

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

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

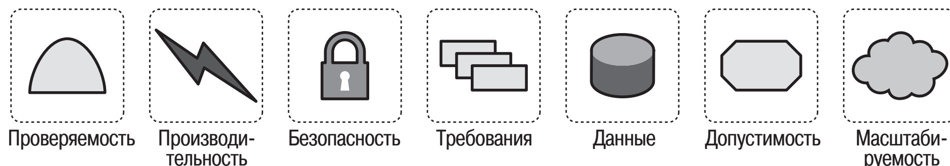

Рис. 1.1. Программная архитектура включает в себя характеристики (всевозможные понятия, заканчивающиеся на «-ость») и требования

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

СИНОНИМЫ ТЕРМИНА «ХАРАКТЕРИСТИКИ АРХИТЕКТУРЫ»

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

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

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

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

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

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

НУЖНА ЛИ АРХИТЕКТУРА AGILE-ПРОЕКТАМ?

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

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

Эволюционная архитектура

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

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

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

Управляемые изменения

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

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

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

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

Инкрементные изменения

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

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

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

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

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

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

Разные измерения архитектуры

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

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

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

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

Техническое измерение

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

Данные

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

Безопасность

Политики и рекомендации по безопасности и инструменты для выявления недостатков.

Операционное/системное измерение

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

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

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

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

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

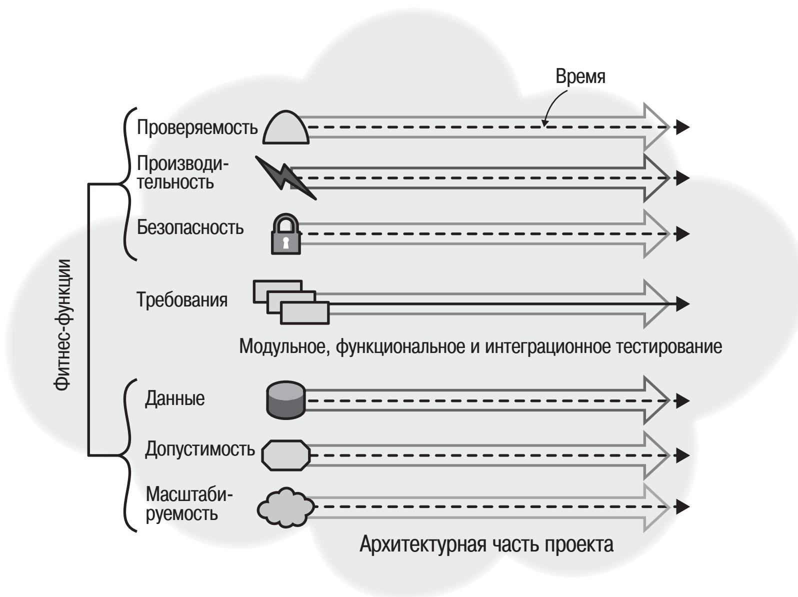

Рис. 1.2. Архитектура включает требования и другие измерения, каждое из которых защищено фитнес-функциями

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

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

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

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

Как осуществлять долгосрочное планирование, если все постоянно меняется?

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

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

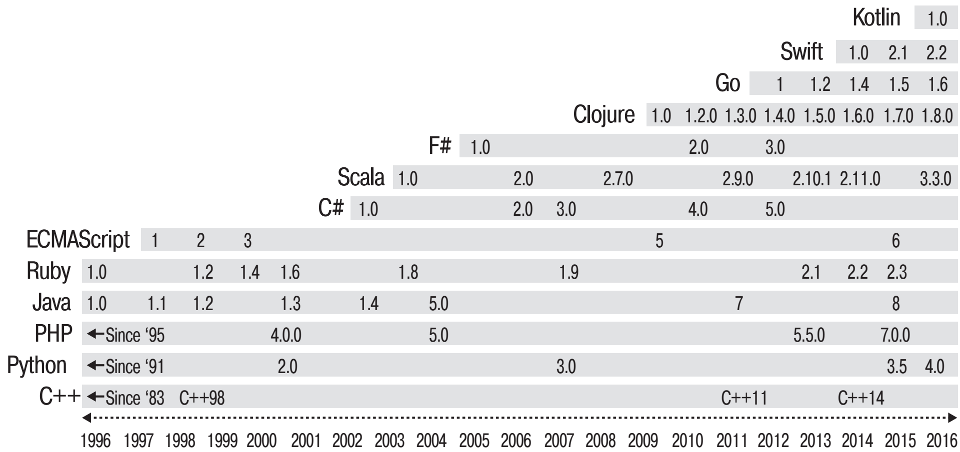

Рис. 1.3. Эволюция популярных языков программирования

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

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

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

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

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

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

Как, построив архитектуру, предотвратить постепенное ухудшение ее качества

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

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

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

Почему архитектура эволюционная

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

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

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

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

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

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

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

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

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

Итоги

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

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