ГЛАВА 1

Что происходит, когда нет «передового опыта»

Почему технические специалисты, такие как архитекторы программного обеспечения, выступают на конференциях или пишут книги? Потому что они открыли для себя то, что в просторечии называется «передовым опытом» (best practices). Однако этим термином настолько злоупотребляют, что те, кто его использует, все чаще реагируют негативно. Но, как бы там ни было, технические специалисты пишут книги, когда придумывают новое решение общей проблемы и хотят донести его до широкой аудитории.

Но что происходит с огромным множеством задач, не имеющих хороших решений? В архитектуре программного обеспечения существуют целые классы задач, которые не имеют универсальных и надежных решений и представляют собой запутанный клубок компромиссов.

Разработчики программного обеспечения приобретают недюжинные навыки поиска в Интернете решений текущей задачи. Например, если нужно выяснить, как настроить конкретный инструмент, то разработчики обращаются за ответом к Google.

Но это не относится к архитекторам.

Для архитекторов многие проблемы являются уникальными, поскольку объединяют среду и обстоятельства в конкретной организации — насколько велики шансы, что кто-то уже сталкивался именно с таким сценарием *и* опубликовал свое решение в блоге или на Stack Overflow?

Архитекторы могут задаваться вопросом, почему так мало книг по архитектуре по сравнению с другими техническими темами, такими как фреймворки, API и т. д. Архитекторы редко сталкиваются с обычными проблемами, им постоянно приходится принимать решения в новых ситуациях. Для архитектора любая задача — снежинка. Во многих случаях задача является новой не только

для конкретной организации, но и для всего мира. По этим задачам не существует ни книг, ни конференций!

Архитекторы не должны постоянно искать серебряные пули — универсальные решения своих задач; эти решения так же редки, как и в 1986 году, когда Фред Брукс (Fred Brooks) ввел этот термин.

> Нет ни одного открытия ни в технологии, ни в методах управления, одно только использование которого обещало бы в течение ближайшего десятилетия на порядок [в десять раз] повысить производительность, надежность, простоту разработки программного обеспечения.
>
> *Фред Брукс, статья No Silver Bullet*

Практически каждая задача привносит новые сложности. Поэтому настоящая работа архитектора заключается в его способности объективно определить возможные компромиссы, оценить их и выбрать хорошее решение. Мы не говорим «лучшее решение» (ни в этой книге, ни в реальной жизни), поскольку «лучшее» подразумевает, что архитектору удалось максимально использовать все конкурирующие факторы в проекте. Вместо этого мы шутливо советуем:

Не пытайтесь найти лучшее решение в архитектуре программного обеспечения; стремитесь к наименее худшему сочетанию компромиссов.

Часто лучшее решение, которое может создать архитектор, представляет собой наименее худший набор компромиссов: никакая архитектурная характеристика не является преобладающей, но баланс всех конкурирующих характеристик способствует успеху проекта.

Напрашивается вопрос: «Как архитектор должен *искать* наименее худшую комбинацию компромиссов (и эффективно документировать их)?» Эта книга в первую очередь посвящена принятию решений и учит архитекторов делать наиболее эффективный выбор в новых ситуациях.

Почему «сложные компромиссы»?

Почему мы назвали эту книгу «Современный подход к программной архитектуре: сложные компромиссы»? На самом деле слово «сложные» в названии выполняет двойную функцию. Во-первых, *«сложный»* означает *«трудный»*, и архитекторы постоянно сталкиваются с трудными задачами, с которыми бук-

вально (и фигурально) никто не сталкивался раньше, включая многочисленные технологические решения, имеющие долгосрочные последствия, наложенные на межличностное и политическое окружение, в котором должно приниматься решение.

Во-вторых, *«сложный»* означает *«неподатливый»*, то есть нечто с трудом поддающееся изменениям и образующее основу для чего-то другого, более податливого, как, например, *аппаратное* обеспечение для *программного*. Точно так же архитекторы обсуждают различия между выстраиванием *архитектуры* и *проектированием*, где первая является структурным понятием, а второе легче изменить. Таким образом, в этой книге мы будем говорить об основополагающих частях архитектуры.

Само определение архитектуры программного обеспечения вызвало многочасовые и бесплодные дискуссии среди практиков. Одно из наших любимых ироничных определений гласит: «Архитектура программного обеспечения — это *такая штука*, которую потом сложно изменить». Этой самой *штуке* и посвящена наша книга.

Советы по архитектуре программного обеспечения, неподвластные времени

Экосистема разработки ПО непрерывно и хаотично растет и меняется. Темы, бывшие насущными несколько лет назад, были либо поглощены экосистемой и исчезли, либо заменены чем-то другим/лучшим. Например, десять лет назад преобладающим архитектурным стилем для крупных предприятий была сервисно-ориентированная архитектура. Теперь его практически никто не использует (по причинам, которые мы раскроем по ходу дела); сегодня предпочтительным стилем организации многих распределенных систем являются микросервисы. Как и почему произошел этот переход?

Рассматривая определенный стиль (особенно бывший актуальным в прошлом), архитекторы должны учитывать ограничения, которые обеспечивали доминирующее положение этого стиля. В то время многие компании объединялись, превращаясь в *корпорации*, и это порождало сопутствующие интеграционные проблемы. Кроме того, крупные компании не считали жизнеспособными решения с открытым исходным кодом (часто по политическим, а не по техническим причинам). Вследствие этого архитекторы сделали упор на общие ресурсы и централизованное управление.

Однако за прошедшие годы открытый исходный код и Linux превратились во вполне жизнеспособные альтернативы, что сделало операционные системы

коммерчески бесплатными. Переломный момент наступил, когда Linux стал *операционно* свободным благодаря появлению таких инструментов, как Puppet и Chef, которые позволили командам разработчиков программно развертывать свои среды в рамках автоматизированной сборки. Эта новая возможность способствовала архитектурной революции микросервисов и быстро развивающейся инфраструктуры контейнеров и инструментов их оркестрации, таких как Kubernetes.

Таким образом, можно увидеть, что экосистема разработки программного обеспечения расширяется и развивается в совершенно неожиданных направлениях. Одна новая возможность ведет к другой, а та неожиданно порождает новые. Со временем экосистема полностью заменяет себя, иногда по частям.

Как следствие, у авторов книг о технологиях в целом и об архитектуре ПО в частности возникает извечная проблема: как написать что-то, что устареет не сразу?

В книге мы не будем фокусироваться на технологиях и других деталях реализации, а сосредоточимся на том, *как* архитекторы принимают решения и как объективно оценивать компромиссы в новых ситуациях. Мы будем использовать современные сценарии и примеры, чтобы предоставить детали и контекст, но основное наше внимание будет сосредоточено на анализе компромиссов и принятии решений при встрече с новыми проблемами.

Важность данных в архитектуре

> Данные — большая ценность, и они будут храниться дольше, чем сами системы.
>
> *Тим Бернерс-Ли (Tim Berners-Lee)*

Для многих архитектур данные — самое главное. Каждое предприятие, создающее какую-либо систему, вынуждено иметь дело с данными, поскольку они, как правило, живут намного дольше, чем системы или архитектура, и требуют тщательного обдумывания и проектирования. Однако склонность архитекторов данных создавать тесно связанные системы приводит к конфликтам в современных распределенных архитектурах. Например, архитекторы и администраторы баз данных (БД) должны обеспечить выживание бизнес-данных после разделения монолитных систем, чтобы бизнес мог по-прежнему извлекать пользу из своих данных независимо от изменений в архитектуре.

Говорят, что *данные — самый важный актив компании*. Предприятия хотят извлекать выгоду из имеющихся у них данных и постоянно ищут новые

способы их использования при принятии решений. Каждое подразделение предприятия теперь управляется данными, от обслуживания существующих клиентов до привлечения новых, удержания клиентов, улучшения продуктов, прогнозирования продаж и других целей. Такая зависимость от данных означает, что вся программная архитектура служит данным, обеспечивая всем подразделениям предприятия доступ к нужным данным и возможность их использования.

За несколько десятилетий авторы книги создали множество распределенных систем, когда они только начали входить в моду. Однако принимать решения в современных микросервисных архитектурах оказалось намного сложнее, и мы хотели выяснить почему. В конечном итоге мы пришли к выводу, что проблема в сохранении всех данных в одной реляционной базе данных — привычке, сложившейся еще перед появлением распределенных архитектур. Однако в микросервисах и предметно-ориентированном проектировании (https://oreil.ly/bW8CH), согласно философии *ограничения контекста* как способа ограничения видимости деталей реализации, данные, вкупе с управлением транзакциями, переместились на архитектурный уровень. Многие из сложных компромиссов современной архитектуры, которым мы уделим внимание в частях I и II, возникают из-за противоречий между данными и архитектурой.

Одно важное различие, которое мы рассматриваем в разных главах, — разделение *операционных* и *аналитических* данных.

- *Операционные данные* используются для ведения бизнеса, включая данные о продажах, сделках, запасах и т. д. На этих данных основана работа компании — если что-то прерывает их поток, то организация долгое время не может нормально функционировать. Этот тип данных определяется как *обработка транзакций в реальном времени* (Online Transactional Processing, OLTP), которая обычно включает добавление, изменение и удаление данных в базе данных.
- *Аналитические данные* используются аналитиками для прогнозирования, определения тенденций и других бизнес-исследований. Эти данные, как правило, не являются транзакционными и часто имеют нереляционный характер. Они могут храниться в графовой базе данных или в моментальных снимках в форме, отличной от исходной транзакционной формы. Эти данные не имеют решающего значения для повседневной работы и обычно используются для долгосрочного планирования и принятия стратегических решений.

В книге мы будем рассматривать влияние обоих видов данных: и операционных, и аналитических.

Запись архитектурных решений

Один из наиболее эффективных способов документирования архитектурных решений — *запись в реестре архитектурных решений* (Architectural Decision Records, ADR (https://adr.github.io/)). Впервые использовать ADR предложил Майкл Найгард (Michael Nygard) в своей статье в блоге (https://oreil.ly/yDcU2), а затем они были отмечены как «принятые» в сборнике трендов/технологий Thoughtworks Technology Radar (https://oreil.ly/0nwHw). ADR состоит из короткого текстового файла (обычно одна-две страницы), описывающего конкретное архитектурное решение. ADR могут быть оформлены как обычные текстовые файлы, но чаще для их оформления применяется какой-либо текстовый формат, такой как AsciiDoc (http://asciidoc.org/) или Markdown (https://www.markdownguide.org/). Кроме того, запись ADR можно оформить с помощью шаблона вики-страницы. Мы посвятили ADR целую главу в нашей предыдущей книге *Fundamentals of Software Architecture*[1] (https://learning.oreilly.com/library/view/fundamentals-of-software/9781492043447).

Мы будем использовать ADR как способ документирования различных архитектурных решений, принимаемых на протяжении всей книги. Для описания каждого архитектурного решения будем использовать следующий формат ADR, предполагая, что каждая запись ADR одобрена.

ADR: короткое именное словосочетание, содержащее архитектурное решение.

Контекст

В этом разделе ADR мы будем приводить краткое описание задачи, состоящее из одного-двух предложений, и перечислять альтернативные решения.

Решение

В этом разделе мы изложим архитектурное решение и его подробное обоснование.

Последствия

В этом разделе ADR мы опишем последствия применения решения, а также обсудим рассмотренные компромиссы.

Список всех записей ADR, созданных в этой книге, можно найти в приложении Б.

Документирование решений важно для архитектора, но управление правильным использованием решения — отдельная тема. К счастью, современные инженерные методы позволяют автоматизировать многие распространенные задачи управления с помощью функций пригодности для архитектуры.

[1] *Форд Н., Ричардс М.* Фундаментальный подход к программной архитектуре.

Функции пригодности

Определив взаимосвязи между компонентами и отразив их в проекте, архитектор должен убедиться, что разработчики будут придерживаться этого проекта. Но как это сделать? Как вообще архитекторы могут гарантировать реализацию определяемых ими архитектурных принципов, если не являются теми, кто этим занимается?

Эти вопросы относятся к категории *управления архитектурой*, применимой к любому организованному надзору за аспектами разработки программного обеспечения. Поскольку эта книга в основном посвящена конструированию архитектуры, мы расскажем, как автоматизировать принципы проектирования и качества с помощью функций пригодности (fitness functions).

Разработка ПО медленно развивалась во времени, адаптируя уникальные инженерные методы. На заре разработки и к крупным процессам (таким как водопадный процесс разработки), и к малым (практики интеграции внутри проектов) обычно применялась метафора производства. В начале 1990-х работы по переоценке инженерных методов разработки ПО, проведенные инженерами проекта C3 под руководством Кента Бека (Kent Beck), привели к созданию методологии экстремального программирования (eXtreme Programming, XP) и показали важность дополнительной обратной связи и автоматизации как ключевых факторов повышения производительности разработчиков. В начале 2000-х те же уроки были применены на стыке разработки и эксплуатации программного обеспечения, благодаря чему родилась новая роль DevOps и были автоматизированы многие рутинные операции, ранее выполняемые вручную. Как и раньше, автоматизация позволяет командам работать быстрее, поскольку им не нужно беспокоиться о том, что что-то пойдет не так и они не получат своевременную обратную связь. Как следствие, *автоматизация* и *обратная связь* стали центральными принципами эффективной разработки ПО.

Рассмотрим среды и ситуации, мешающие автоматизации. До появления непрерывной интеграции большинство программных проектов предусматривало продолжительный этап интеграции. Предполагалось, что каждый разработчик будет работать до определенной степени изолированно от других, а затем все вместе они будут объединять код на этапе интеграции. Следы этой практики все еще сохраняются в инструментах управления версиями, которые вызывают ветвление и препятствуют непрерывной интеграции. Неудивительно, что существовала сильная корреляция между размером проекта и сложностью этапа интеграции. Внедрив непрерывную интеграцию, команда, практикующая экстремальное программирование (XP), продемонстрировала ценность быстрой и непрерывной обратной связи.

Революция DevOps пошла по тому же пути. Linux и другое ПО с открытым исходным кодом становилось «достаточно хорошим» для предприятий. Вдобавок появлялись инструменты, позволявшие программно определять (в конечном итоге) виртуальные машины. По мере всего этого операционный персонал понял, что может автоматизировать определение машин и многие другие повторяющиеся задачи.

В обоих случаях достижения в области технологий и знаний привели к автоматизации повторяющихся задач, решение которых прежде обходилось довольно дорого, что описывает текущее состояние управления архитектурой в большинстве организаций. Например, выбрав определенный архитектурный стиль или средство коммуникации, как архитектор сможет убедиться, что разработчик правильно его реализует? Когда все делается вручную, архитекторы проводят обзоры кода или собирают комиссию для анализа архитектуры, чтобы получить оценку состояния управления. Однако, как и при ручной настройке компьютеров в процессе эксплуатации, при поверхностном рассмотрении важные детали легко могут остаться незамеченными.

Использование функций пригодности. В книге *Building Evolutionary Architectures*[1] (O'Reilly) 2017 года авторы (Нил Форд, Ребекка Парсонс и Патрик Куа) определили концепцию *функции пригодности для архитектуры*: любой механизм, объективно оценивающий целостность некой архитектурной характеристики или их комбинации. Коротко разберем это определение по пунктам.

- *Любой механизм*. Архитекторы могут реализовывать функции пригодности с помощью широкого спектра инструментов; в этой книге мы покажем множество примеров. Например, существуют специальные библиотеки тестирования, позволяющие проверять структуру архитектуры; с помощью мониторов архитекторы могут тестировать операционные характеристики архитектуры, такие как производительность или масштабируемость, а фреймворки хаос-инжиниринга служат для проверки надежности и отказоустойчивости.
- *Объективная оценка целостности*. Один из ключевых факторов автоматизированного управления — объективное определение характеристик архитектуры. Например, архитектор не может просто сказать, что ему нужен «высокопроизводительный» сайт; он должен представить величину, которую можно измерить с помощью теста, монитора или другой функции пригодности.

  Архитекторы должны следить за *составными архитектурными характеристиками*, не поддающимися объективному измерению и являющимися

[1] *Форд Н., Парсонс Р., Куа П.* Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2022.

составными частями других измеримых показателей. Например, «гибкость» не поддается измерению. Но если разобрать широкий термин *«гибкость»* с точки зрения архитектора, то оказывается, что цель состоит в быстром и уверенном реагировании команды на изменения в экосистеме или предметной области. Соответственно, архитектор может найти измеримые характеристики, отражающие степень гибкости: возможность развертывания, тестируемость, продолжительность цикла разработки и т. д. Часто отсутствие возможности измерить архитектурную характеристику указывает на слишком расплывчатое определение. Стремление найти измеримые свойства позволяет архитекторам автоматизировать применение функций пригодности.

- *Некая архитектурная характеристика или их комбинация.* Архитектурная характеристика описывает два вида функций пригодности:
  - *атомарные* оценивают единственную архитектурную характеристику. Например, функция пригодности, проверяющая наличие циклических зависимостей компонентов в кодовой базе, является атомарной;
  - *комплексные* проверяют комбинацию архитектурных характеристик. Усложняющей особенностью архитектурных характеристик является синергия с другими характеристиками, которую они иногда демонстрируют. Например, если архитектор хочет улучшить безопасность, то велика вероятность, что это повлияет на производительность. Точно так же масштабируемость и адаптируемость иногда противоречат друг другу — поддержка большого количества одновременно работающих пользователей может затруднить обработку внезапных всплесков. Комплексные функции пригодности проверяют комбинацию взаимосвязанных архитектурных характеристик и оценивают негативное влияние комбинированного эффекта на архитектуру.

Архитектор реализует функции пригодности в целях защиты от неожиданных изменений архитектурных характеристик. В мире гибкой разработки программного обеспечения разработчики реализуют модульные и функциональные тесты, а пользователи — приемочные, чтобы проверить различные *предметные* аспекты. Однако до сих пор не существовало подобного механизма, проверяющего *архитектурные* свойства проекта. На самом деле разделение между функциями пригодности и модульными тестами — хороший ориентир для архитекторов. Функции пригодности проверяют архитектурные свойства, а не предметные; модульные тесты делают обратное. Соответственно, архитектор может определить, что именно нужно, функция пригодности или модульный тест, задав вопрос: «Требуются ли какие-либо знания предметной области для этой проверки?» Если ответ «да», то уместно модульное/функциональное/приемочное тестирование; если «нет», то нужна функция пригодности.