Краткое содержание

Оглавление

ЧАСТЬ II. ОБЪЕДИНЯЕМ ВСЕ ВМЕСТЕ

ПРИЛОЖЕНИЯ

Отзывы о книге «Современный подход к программной архитектуре: сложные компромиссы»

Эта книга — долгожданное руководство по созданию микросервисов и анализу нюансов архитектурных решений по всему стеку технологий. Она может служить каталогом архитектурных решений, которые могут приниматься при построении распределенной системы, и описывает плюсы и минусы каждого решения. Ее должен прочитать каждый архитектор, занимающийся созданием современных распределенных систем.

Александар Серафимоски (Aleksandar Serafimoski), ведущий консультант, Thoughtworks

Бесценная книга для технологов, увлеченных разработкой архитектур. Содержит превосходное описание паттернов.

Ванья Сетх (Vanya Seth), руководитель технического отдела, Thoughtworks, Индия

Кем бы вы ни были — начинающим архитектором или опытным руководителем команды, эта книга, обойдясь без словоблудия, расскажет вам, как добиться успеха на пути к созданию корпоративных приложений и микросервисов.

Д-р Венкат Субраманиам (Venkat Subramaniam), обладатель множества наград, автор и основатель Agile Developer, Inc.

Эта книга содержит ценную информацию, практические советы и примеры из реальной жизни, посвященные разделению тесно связанных систем и их повторному созданию. Приобретя навыки эффективного анализа компромиссов, вы начнете принимать более эффективные архитектурные решения.

Йост ван Винен (Joost van Weenen), управляющий партнер и сооснователь Infuze Consulting

Мне очень понравился этот всеобъемлющий труд по распределенным архитектурам! Отличное сочетание серьезного обсуждения, фундаментальных идей и множества практических советов.

Дэвид Клоэт (David Kloet), независимый архитектор программного обеспечения

Разбить большой ком грязи — непростая работа. Начиная с кода и заканчивая данными, эта книга поможет увидеть, какие сервисы следует отделить и сделать независимыми, а какие оставить работать вместе.

Рубен Диас-Мартинес (Rubén Díaz-Martínez), разработчик программного обеспечения в Codesai

Эта книга поможет вам заложить теоретическую и практическую основы и ответить на самые сложные вопросы, возникающие при разработке современных архитектур программного обеспечения.

Джеймс Льюис (James Lewis), технический директор, Thoughtworks

Вступление

Работая над книгой *Fundamentals of Software Architecture*[1], мы (Нил и Марк) постоянно сталкивались со сложными примерами архитектур, о которых нам очень хотелось рассказать, но которые были слишком трудными. Ни один из них не имел простого решения, и каждый являл собой клубок запутанных компромиссов. Мы сложили эти примеры в стопку, которую назвали «Сложные компромиссы». Как только та книга была закончена, мы посмотрели на эту огромную стопку и попытались выяснить: *почему эти проблемы так трудно решить в современных архитектурах*?

Мы проработали все примеры как архитекторы, применяя анализ компромиссов для каждой ситуации, а также обращая внимание на процесс, с помощью которого достигались компромиссы. Одним из наших первых открытий стало возрастающее значение данных в архитектурных решениях: кто может/должен иметь доступ к данным, кто может/должен формировать и изменять их и как управлять разделением аналитических и операционных данных. Чтобы это понять, мы пригласили экспертов в указанных областях, что позволило увидеть весь процесс принятия решений с обеих сторон: от архитектуры к данным и от данных к архитектуре.

Результатом стала эта книга: сборник сложных проблем в современной архитектуре программного обеспечения, компромиссов, усложняющих принятие решений, и, наконец, иллюстрированное руководство, показывающее, как применять тот же анализ компромиссов к вашим собственным уникальным задачам.

Условные обозначения

В этой книге приняты следующие обозначения.

Курсив

Курсивом выделены новые термины и ключевые понятия.

---

[1] *Форд Н., Ричардс М.* Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы. — СПб.: Питер, 2023.

`Моноширинный шрифт`

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

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсив`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот рисунок указывает на совет или предложение.

Использование программного кода примеров

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

Если у вас есть вопросы технического характера или возникли проблемы с использованием примеров кода, то присылайте их по адресу bookquestions@oreilly.com.

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

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

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

Благодарности

Мы, Марк и Нил, выражаем благодарность всем, кто посещал наши (почти исключительно онлайн) занятия, семинары, конференции и встречи групп пользователей, а также всем, кто выслушивал наши доклады по данной теме и оставлял бесценные отзывы. Многократное повторное обсуждение нового материала — особенно сложная задача, когда нельзя сделать это вживую, поэтому мы очень ценим усилия всех, кто оставался с нами на протяжении множества итераций. Мы благодарим команду издательства O'Reilly, сделавшую процесс написания книги настолько безболезненным, насколько это возможно. Мы также благодарим отдельные группы, такие как Pasty Geeks и Hacker B&B, являющиеся оазисами здравомыслия и источниками искрометных идей.

Спасибо всем рецензентам нашей книги: Ванье Сетху (Vanya Seth), Венкату Субраманиану (Venkat Subramanian), Йосту ван Винену (Joost van Weenen), Грэди Бучу (Grady Booch), Рубену Диасу (Ruben Diaz), Дэвиду Клоэту (David Kloet), Мэтту Штейну (Matt Stein), Данило Сато (Danilo Sato), Джеймсу Льюису (James Lewis) и Сэму Ньюману (Sam Newman). Ваши знания и отзывы помогли улучшить эту книгу.

Мы хотим выразить особую признательность многим работникам и семьям, пострадавшим от неожиданной глобальной пандемии. Как работники умственного труда, мы столкнулись с неудобствами, которые не выдерживают никакого сравнения с массовыми потрясениями и разрушениями во всех сферах жизни, причиненными многим нашим друзьям и коллегам. Мы особенно сочувствуем и признательны работникам здравоохранения, многие из которых не думали, что когда-нибудь окажутся на переднем крае ужасной глобальной трагедии, но достойно справились с ней. Наша благодарность вам бесконечна.

От Марка Ричардса

В дополнение к написанному выше я хочу еще раз поблагодарить мою прекрасную супругу Ребекку (Rebecca) за то, что она терпела, пока я занимался еще одним книжным проектом. Твоя неиссякаемая поддержка и советы помогли мне написать эту книгу, и это притом, что тебе приходилось отвлекаться от работы

над собственным романом. Ребекка, ты для меня — целый мир. Я также благодарю моего доброго друга и соавтора Нила Форда. Совместная работа с тобой над этой книгой (и над предыдущей) была по-настоящему ценным и полезным опытом. Ты есть и всегда будешь моим другом.

От Нила Форда

Я хотел бы поблагодарить свою большую семью: весь коллектив Thoughtworks, а также Ребекку Парсонс (Rebecca Parsons) и Мартина Фаулера (Martin Fowler). Thoughtworks — выдающаяся группа людей, которые создают для клиентов ценные решения, внимательно следят за их работой и выясняют, что еще в них можно улучшить. Компания Thoughtworks во многом поддержала эту книгу и продолжает растить мыслителей, бросающих мне вызов и вдохновляющих меня каждый день. Я также благодарю наш коктейль-клуб, находящийся по соседству, за регулярную возможность сбежать от рутины и в том числе за еженедельные встречи на свежем воздухе, этакие социально дистанцированные версии, которые помогли всем нам пережить пандемию. Я благодарю своего давнего друга Нормана Запиена (Norman Zapien), который постоянно дарит мне приятные беседы. Наконец, спасибо моей жене Кэнди (Candy), благодаря которой у меня есть возможность тратить много времени на такие занятия, как написание книг, а не на уход за нашими кошками.

От Прамода Садаладжа

Я благодарю свою супругу Рупали (Rupali) за поддержку и понимание и моих прекрасных девочек Арулу (Arula) и Архану (Arhana) за воодушевление; папа любит вас обеих. Все, что я делаю, было бы невозможно без клиентов, с которыми я работаю, и различных конференций, помогающих мне проверять и пересматривать идеи и понятия. Я благодарю AvidXchange, последнего клиента, с которым я работаю, за его поддержку и предоставленную возможность проверки новых концепций. Я также благодарю Thoughtworks за постоянную поддержку, а также Нила Форда (Neal Ford), Ребекку Парсонс (Rebecca Parsons) и Мартина Фаулера (Martin Fowler), ставших мне замечательными наставниками; вы помогаете мне стать лучше. Наконец, спасибо моим родителям, особенно моей маме Шобхе (Shobha), по которой я постоянно скучаю. *Я очень скучаю по тебе, мама*.

От Жамак Дехгани

Я благодарю Марка и Нила за их приглашение внести свой вклад в эту удивительную книгу. Но это было бы невозможно без постоянной поддержки моего мужа Адриана (Adrian) и терпения моей дочери Арианны (Arianna). Я люблю вас обоих.

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.

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.

Например, говоря об *адаптируемости*, архитекторы подразумевают способность приложения выдерживать внезапный наплыв пользователей. Обратите внимание, что в этом случае архитектору не нужно знать никаких подробностей о предметной области — это может быть сайт электронной коммерции, онлайн-игра или что-то еще. Поэтому *адаптируемость* является архитектурной характеристикой, и для ее оценки требуется функция пригодности. Напротив, проверять правильность элементов адреса электронной почты нужно с помощью традиционного теста. Конечно, такое разделение не является четким и ясным — некоторые функции пригодности будут касаться предметной области и наоборот, но осознание разных целей помогает мысленно разделить их.

Приведем несколько примеров, чтобы сделать обсуждение менее абстрактным.

Одна из типичных целей архитектора — обеспечить хорошую внутреннюю структурную целостность кодовой базы. Однако на многих платформах злые силы препятствуют добрым намерениям архитектора. Например, в любой популярной среде разработки для Java или .NET, как только разработчик ссылается на класс, который еще не импортирован, IDE услужливо выводит диалоговое окно, в котором разработчику предлагается автоматически импортировать ссылку. Это происходит так часто, что у большинства программистов вырабатывается рефлекс отвечать согласием.

Однако произвольное импортирование классов или компонентов друг из друга означает катастрофу для модульности. Например, на рис. 1.1 показан особенно опасный антипаттерн, которого архитекторы стремятся избежать.

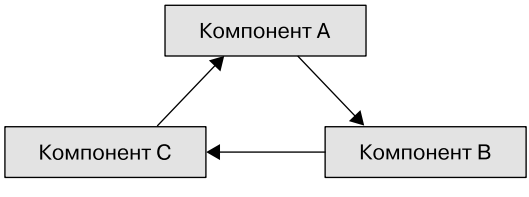

Рис. 1.1. Циклические зависимости между компонентами

В этом антипаттерне каждый компонент ссылается на два других. Такая сеть компонентов вредит модульности, поскольку разработчик не может повторно использовать один компонент, не привлекая другие. И конечно же, если другие компоненты связаны еще с какими-то, архитектура все больше и больше склоняется к антипаттерну Большой ком грязи (https://oreil.ly/usx7p). Могут ли архитекторы управлять этим поведением, не контролируя постоянно разработчиков, обожающих автоматический импорт зависимостей? Обзоры кода могут помочь,

но выполняются слишком поздно в цикле разработки, чтобы быть эффективной мерой. Если архитектор позволит команде разработчиков безудержно импортировать зависимости из кодовой базы в течение недели до следующего обзора кода, то базе будет нанесен серьезный ущерб.

Решение этой проблемы состоит в том, чтобы написать функцию пригодности, проверяющую наличие циклических зависимостей, как показано в примере 1.1.

Пример 1.1. Функция пригодности для определения циклических зависимостей между компонентами

```
public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
    void init() {
          jdepend = new JDepend();
          jdepend.addDirectory("/path/to/project/persistence/classes");
          jdepend.addDirectory("/path/to/project/web/classes");
          jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
          Collection packages = jdepend.analyze();
          assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}
```

Здесь архитектор использовал инструмент метрик JDepend (https://oreil.ly/ozzzk), проверяющий зависимости между пакетами. Инструмент исследует структуру пакетов Java, и тест терпит неудачу, если обнаружатся какие-либо циклические зависимости. Архитектор может включить этот тест в процесс непрерывной сборки проекта и перестать беспокоиться о том, что разработчики случайно создадут циклы. Это отличный пример функции пригодности, охраняющей значимые методы разработки программного обеспечения: это важная задача для архитекторов, но она мало влияет на повседневную практику.

В примере 1.1 показана весьма низкоуровневая функция пригодности, ориентированная на код. Есть множество популярных инструментов гигиены кода (таких как SonarQube (https://www.sonarqube.org/)), которые реализуют многие типовые функции пригодности «под ключ». Однако архитектору может понадобиться проверить также макроструктуру архитектуры. Проектируя многоуровневую архитектуру (такую как на рис. 1.2), архитектор определяет уровни, чтобы обеспечить разделение задач.

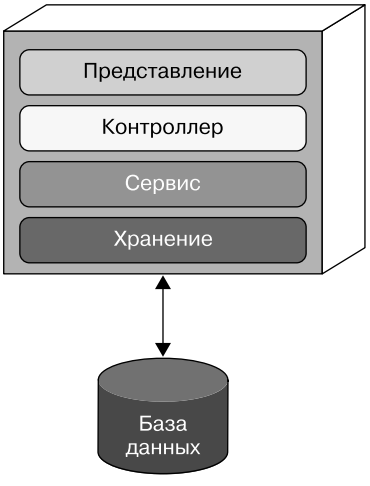

Рис. 1.2. Традиционная многоуровневая архитектура

Как архитектор сможет проверить, что разработчики соблюдают границы уровней? Одни разработчики могут не понимать важности паттернов, тогда как другие могут занимать позицию «лучше попросить прощения, чем разрешения», стремясь поскорее решить какую-то локальную задачу, например повысить производительность. Но если разработчикам позволено нарушать границы, то долгосрочному состоянию архитектуры наносится вред.

ArchUnit (https://www.archunit.org/) позволяет архитекторам решить эту проблему с помощью функции пригодности, показанной в примере 1.2.

Пример 1.2. Функция ArchUnit, служащая для проверки соблюдения границ уровней

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

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

В примере 1.2 архитектор определяет желательную взаимосвязь между уровнями и пишет функцию пригодности, предназначенную для проверки. Это позволяет архитектору устанавливать принципы архитектуры, помимо диаграмм и других информационных артефактов, и постоянно проверять их.

Аналогичный инструмент в пространстве .NET, NetArchTest (https://oreil.ly/EMXpv), дает возможность выполнять такие же проверки для этой платформы. Проверка границ уровней в C# показана в примере 1.3.

Пример 1.3. Использование NetArchTest для проверки границ уровней

```
// Классы, определенные на уровне представления,
// не должны напрямую обращаться к репозиториям
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

В данном пространстве постоянно появляются все более сложные инструменты. Мы продолжим освещать эти методы, иллюстрируя функции пригодности наряду со многими нашими решениями.

Поиск объективного результата для функции пригодности имеет решающее значение. Однако *объективность* не означает *статичность*. Одни функции пригодности будут иметь неконтекстные возвращаемые значения, такие как `true/false`, или числовые значения, такие как порог производительности. Другие (считающиеся *динамическими*) будут возвращать значения, основанные на некоем контексте. Например, при измерении *масштабируемости* архитекторы оценивают количество одновременно работающих пользователей, а также производительность для каждого пользователя. Часто архитекторы проектируют системы так, чтобы при увеличении количества пользователей производительность на одного пользователя немного снижалась, но не падала резко. Для таких систем архитекторы разрабатывают функции пригодности, оценивающие производительность, которые учитывают количество одновременно работающих пользователей. Пока оценка архитектурной характеристики объективна, архитекторы могут ее тестировать.

Применение большинства функций пригодности можно и нужно автоматизировать, чтобы их можно было выполнять постоянно. Но будут и те, которые придется применять вручную. Такая функция пригодности требует, чтобы проверкой занимался человек. Например, при работе с системой, оперирующей конфиденциальной юридической информацией, юристу в целях обеспечения безопасности может потребоваться проверить изменения в критических частях, а это нельзя автоматизировать. Большинство конвейеров развертывания поддерживают этапы ручной проверки, что позволяет командам использовать ручные функции пригодности. В идеале они должны запускаться настолько часто, насколько это возможно, — проверка, которая не выполняется, ничего не сможет

проверить. Команды выполняют функции пригодности по запросу (редко) или в рамках процесса непрерывной интеграции (чаще всего). Чтобы в полной мере воспользоваться преимуществами проверок, которые производятся функциями пригодности, их следует запускать постоянно.

Итак, непрерывность важна, как было показано в примере использования функций пригодности. Рассмотрим следующий сценарий: что делает компания, когда в одной из сред разработки или в какой-то библиотеке, используемых предприятием, обнаруживается эксплойт «нулевого дня»? Как и в большинстве компаний, эксперты по безопасности просматривают проекты, чтобы найти проблемную версию фреймворка и выполнить обновление, но этот процесс редко автоматизируется и основывается на множестве ручных операций. Это не абстрактный вопрос; именно этот сценарий касался крупного финансового учреждения (см. ниже врезку «Утечка данных в Equifax»). Как и при управлении архитектурой, описанном ранее, в ходе ручных процессов возникают ошибки и упускаются детали.

УТЕЧКА ДАННЫХ В EQUIFAX

Седьмого сентября 2017 года Equifax, крупнейшее в США агентство по проверке кредитоспособности, объявило об утечке данных. Проблема была связана со взломом популярного веб-фреймворка Struts в экосистеме Java (Apache Struts vCVE-2017-5638). Организация-разработчик опубликовала заявление об уязвимости и выпустила исправление 7 марта 2017 года. На следующий день Министерство внутренней безопасности связалось с Equifax и аналогичными компаниями, предупредив о проблеме, и 15 марта 2017 года они провели сканирование, в ходе которого обнаружились не все уязвимые системы. В результате критическое исправление не было применено ко многим старым системам до 29 июля 2017 года, когда эксперты по безопасности Equifax выявили вторжение, приведшее к утечке данных.

Представьте альтернативный мир, в котором каждый проект имеет конвейер развертывания, а у группы безопасности есть свой «слот» в конвейере развертывания каждой команды, где они могут развертывать функции пригодности. В большинстве случаев это будут рутинные проверки мер безопасности, такие как предотвращение хранения паролей в базах данных и аналогичные обычные задачи по управлению. Однако при появлении эксплойта «нулевого дня» наличие везде одного и того же механизма позволит команде безопасности вставить тест в каждый проект, который проверит наличие определенного фреймворка и номер его версии и при обнаружении опасной версии завершит сборку, отправив уведомление группе безопасности. Команды настраивают конвейеры развертывания так, чтобы они начинали работу при появлении любых изменений в экосистеме: коде, схеме базы данных, конфигурации развертывания и функциях пригодности. Это позволяет предприятиям повсеместно автоматизировать важные задачи управления.

Функции пригодности дают архитекторам массу преимуществ, не последним из которых является возможность снова заняться программированием! Архитекторы часто жалуются, что они больше не пишут код, но функции пригодности часто являются кодом! Создавая выполняемую спецификацию архитектуры, которую каждый сможет проверить в любое время, запустив сборку проекта, архитекторы должны хорошо понимать систему и ее текущее развитие. Все это пересекается с основной целью — не отставать от кода проекта по мере его роста.

Какими бы эффективными ни были функции пригодности, архитекторам следует избегать чрезмерного их использования. Архитекторы не должны объединяться в клику и уединяться в башне из слоновой кости, чтобы построить невероятно сложный, взаимосвязанный набор функций пригодности, вызывающих раздражение у разработчиков и команды. Архитекторы должны рассматривать это как способ создания выполняемого контрольного списка *важных*, но не *срочных* аспектов программных проектов. Многие проекты являются срочными, из-за чего некоторые важные принципы выпадают из внимания. Это частая причина появления технического долга: «Мы знаем, что это плохо, но исправим это позже», а позже так и не наступает. Воплощая правила, которые касаются качества кода, структуры и других мер предосторожности в функции пригодности и выполняются постоянно, архитекторы создают чек-лист (контрольный список) качества, который разработчики не смогут пропустить.

Несколько лет назад в превосходной книге Атула Гаванде (Atul Gawande (Picador)) *The Checklist Manifesto*[1] освещалось использование чек-листов профессионалами: хирургами, пилотами гражданской авиации и представителями других профессий, которые обычно (а иногда в силу закона) используют контрольные списки как часть своей работы, но совсем не потому, что не знают своей работы или слишком забывчивы. Когда профессионалы выполняют одну и ту же задачу снова и снова, становится легко обмануть себя и случайно пропустить какое-то важное действие, а чек-листы предотвращают это. Функции пригодности являются чек-листом важных принципов, определяемых архитекторами, и запускаются как часть конвейера сборки, чтобы гарантировать, что разработчики случайно (или намеренно, из-за внешних обстоятельств, таких как нехватка времени) не пропустят их.

Мы будем использовать функции пригодности на протяжении всей книги, когда появится возможность проиллюстрировать управление архитектурным решением и первоначальным проектом.

---

[1] *Гаванде А.* Чек-лист. Как избежать глупых ошибок, ведущих к фатальным последствиям.