1 Введение

В этой главе:

- ✓ Важные компромиссы в эксплуатируемых системах.
- ✓ Последствия модульного и интеграционного тестирования.
- ✓ Паттерны проектирования и программирования не решают всех проблем.

При проектировании кода, API и системных архитектур приходится принимать решения, влияющие на обслуживание, производительность, расширяемость и многие другие факторы. Почти всегда решение пойти в одном направлении ограничивает возможность развиваться в другом. Чем дольше живут системы, тем сложнее изменить их архитектуру и отказаться от предыдущих решений. Компромиссы в проектировании и программировании, представленные в этой книге, связаны с выбором из двух и более направлений, в которых может развиваться система. Важно понимать, что, какой бы вариант вы ни выбрали, придется иметь дело со всеми его плюсами и минусами.

Команда должна принимать эти непростые решения в зависимости от контекста, времени выхода на рынок, соглашения об уровне обслуживания (SLA, Service-Level Agreement) и других факторов. Мы опишем некоторые компромиссы, на которые приходится идти в рабочих системах, и сравним их с альтернативными подходами. Надеемся, что после прочтения этой книги вы научитесь внимательно оценивать свои ежедневные решения в проектировании. Это позволит вам принимать их осознанно, учитывая все «за» и «против».

Первая часть книги посвящена низкоуровневым проектным решениям, которые вынужден принимать каждый программист при работе с кодом и API. Вторая часть освещает системы более широко — в ней рассматривается архитектура и потоки данных между компонентами. Также мы поговорим о компромиссах, на которые приходится идти при работе с распределенными системами.

В следующих разделах продемонстрирован подход к анализу компромиссов, принятый в нашей книге. Сначала мы сосредоточимся на компромиссах, с которыми сталкивается любой разработчик: балансе между модульными, интеграционными, сквозными и другими видами тестов. В реальных условиях программный продукт приносит пользу в течение ограниченного периода. Поэтому нужно решить, на какие тесты выделить больше времени — модульные, интеграционные, сквозные или другие. Мы проанализируем плюсы и минусы увеличения количества тестов конкретного типа.

Затем мы рассмотрим зарекомендовавший себя паттерн Одиночка (Singleton) и объясним, как его полезность меняется в зависимости от контекста (однопоточного или многопоточного). Наконец, рассмотрим архитектурные компромиссы более высокого уровня: выбор между микросервисной и монолитной архитектурой.

Следует заметить, что зачастую архитектуру невозможно описать как чисто монолитную или чисто микросервисную. На практике распространен гибридный подход: часть функционала реализована в виде сервисов, тогда как другие части системы могут существовать в монолитном виде. Например, когда унаследованная система монолитна и лишь малая ее часть перемещена в микросервисную архитектуру. Также в проекте, создаваемом с нуля, вполне разумно начать с одного приложения и не разбивать его на микросервисы, если это сопряжено с ощутимыми затратами. Мы кратко проанализируем компромиссы между микросервисами и монолитами. Рекомендуем частично применять эту аргументацию в актуальном контексте, даже если это гибридная архитектура.

В следующих разделах представлен подход, который используется во всех главах: решение задачи в конкретной ситуации, затем анализ альтернативного варианта и, наконец, добавление контекста, подразумевающего компромиссы и окончательные решения. Плюсы и минусы каждого из них проанализированы в конкретном контексте. В следующих главах компромиссные решения будут рассмотрены более подробно.

1.1. ПОСЛЕДСТВИЯ КАЖДОГО РЕШЕНИЯ И ПАТТЕРНА

Цель книги — продемонстрировать различные компромиссы и ошибки в проектировании и программировании. При разборе проектных решений и компромиссов

я буду исходить из того, что вы пишете достаточно хороший код. Когда качество кода становится приемлемым, необходимо определить направление, в котором он должен развиваться.

Чтобы понять логику каждой главы, начнем с изучения компромиссов между двумя самыми полезными и очевидными приемами тестирования, которые должны применяться в коде: интеграционными и модульными тестами. Главная цель — обеспечить покрытие практически каждого пути тестами этих двух видов. Зачастую на практике это нецелесообразно, потому что время, выделенное для написания и тестирования кода, ограниченно. Следовательно, выбор соотношения модульного и интеграционного тестирования — это стандартный компромисс, на который придется пойти.

1.1.1. Решения в модульном тестировании

При написании тестов необходимо решить, какую часть кода тестировать. Возьмем простой компонент, для которого нужно организовать модульное тестирование. Допустим, компонент `SystemComponent` предоставляет один открытый метод API: `publicApiMethod()`. Другие методы скрываются от клиентов приватным модификатором доступа. В следующем листинге приведен код этого сценария.

Листинг 1.1. Компонент для модульного тестирования

```
public class SystemComponent {

  public int publicApiMethod() {
    return privateApiMethod();
  }

  private int privateApiMethod() {
    return complexCalculations();
  }

  private int complexCalculations() {
    // Сложная логика
    return 0;
  }
}
```

Здесь мы должны решить, стоит ли включать в модульное тестирование `complexCalculations()` или же оставить приватный метод скрытым. Такой модульный тест работает по принципу «черного ящика» и покрывает только открытый API. Обычно этого достаточно. Но иногда приватные методы содержат сложную логику, которую также стоит протестировать. В таких ситуациях можно понизить модификатор доступа `complexCalculations()`. Этот подход продемонстрирован в следующем листинге.

Листинг 1.2. Открытый уровень видимости компонента для модульного тестирования

```
@VisibleForTesting
public int complexCalculations() {
  // Сложная логика
  return 0;
}
```

Переходя на открытый уровень видимости, вы разрешаете себе написать модульный тест для части API, которая не должна быть открытой. Открытый метод будет видимым для клиентов API, и появится риск того, что они будут напрямую использовать этот API. В листинге аннотация `@VisibleForTesting` (см. http://mng.bz/y4wq) служит только для информационных целей. Ничто не мешает пользователям вызвать открытый метод API. Если они не заметят эту аннотацию, то могут проигнорировать ее.

Вы можете использовать любой из двух подходов к модульному тестированию, рассмотренных в этом разделе. Последний обеспечивает большую гибкость, но затраты на обслуживание могут возрасти. Возможно, вы выберете промежуточное решение. Для этого можно сделать код пакетно-приватным. Иначе говоря, когда тесты находятся в одном пакете с рабочим кодом, делать его открытым не нужно, но можно использовать эти методы в тестовом коде.

1.1.2. Соотношение модульных и интеграционных тестов

При тестировании логики нужно определиться с соотношением интеграционных и модульных тестов в системе. Часто выбор одного направления снижает возможность развиваться в другом. Кроме того, такое ограничение может зависеть от времени начала разработки системы.

Так как время разработки функционала обычно ограниченно, нужно выбрать, чему посвятить большую его часть — модульным или интеграционным тестам. Тестирование реальных систем должно включать обе эти разновидности, и мы должны определить их соотношение.

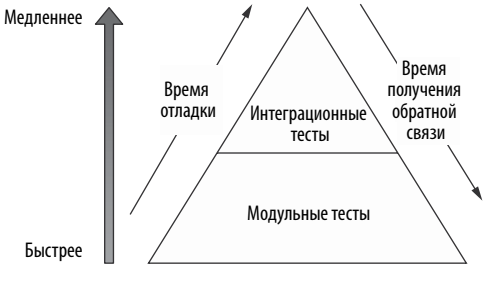

Рис. 1.1. Интеграционные/модульные тесты и время (скорость), необходимое для их выполнения

У каждого вида тестов есть достоинства и недостатки, и выбор между ними становится типичным компромиссом при написании кода. Модульные тесты работают быстрее и обеспечивают более быстрый отклик, так что процесс отладки часто ускоряется. На рис. 1.1 представлены достоинства и недостатки обоих тестов.

Диаграмма на рис. 1.1 имеет форму пирамиды, потому что обычно в программных системах больше модульных тестов, чем интеграционных. Модульные тесты обеспечивают практически мгновенную обратную связь для разработчика, что повышает производительность. Кроме того, они быстрее выполняются и сокращают время отладки кода. Если обеспечить полное покрытие кодовой базы модульными тестами, то при появлении новой ошибки проблема с большой вероятностью будет перехвачена одним из них. Ее можно будет обнаружить на уровне метода, который покрывается конкретным модульным тестом.

С другой стороны, если в системе отсутствуют интеграционные тесты, исчезает возможность анализа связей между компонентами и их объединением в систему. У вас будут хорошо протестированные алгоритмы, но без проверки общей картины. В результате может получиться система, которая правильно функционирует на низком уровне, но без тестирования компонентов невозможно оценить ее работу на более высоком уровне. В реальной жизни код должен сочетать модульные и интеграционные тесты.

Важно заметить, что рис. 1.1 учитывает только один аспект тестирования: время выполнения и, следовательно, время получения обратной связи. В реально эксплуатируемых системах существуют другие уровни тестирования. В них могут быть сквозные тесты, которые осуществляют комплексную проверку бизнес-сценариев. Возможно, в более сложных архитектурах для обеспечения этой бизнес-функциональности придется запустить *N* сервисов, взаимодействующих друг с другом. У таких тестов обычно медленное время отклика из-за лишних затрат ресурсов на подготовку тестовой инфраструктуры. С другой стороны, они дают более высокую уверенность относительно сквозной логики и правильности системы. Сравнение таких тестов с модульными и интеграционными тестами можно производить по разным параметрам — например, насколько хорошо они проверяют систему в целом, как показано на рис. 1.2.

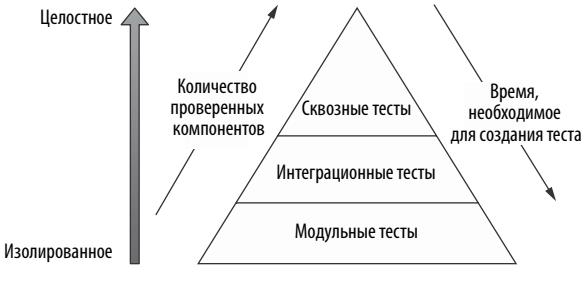

Рис. 1.2. Интеграционные/модульные тесты и время (скорость), необходимое для их выполнения

Так как модульные тесты выполняются изолированно, они не дают сколько-нибудь значимой информации о других компонентах системы и их взаимодействии между собой. Интеграционные тесты пытаются проверять другие компоненты и взаимодействия между ними. Но, как правило, они не охватывают многочисленые (микро-) сервисы, обеспечивающие заданную бизнес-функциональность. Наконец, хотя сквозные тесты проверяют целостность работы системы, количество проверяемых компонентов может быть значительным, так как придется запускать всю инфраструктуру, которая может включать *N* микросервисов, баз данных, очередей и т. д.

Другое измерение (ресурс), которое необходимо принять во внимание, — время на создание тестов. За короткое время можно создать много модульных тестов, так как они относительно просты в разработке. На проектирование интеграционных тестов часто уходит больше времени. Наконец, сквозные тесты требуют значительных начальных вложений для построения необходимой для них инфраструктуры.

На практике ресурсы ограничены (например, бюджет и время), и приходится максимизировать качество программного продукта с учетом этих условий. Однако покрытие кода тестами позволяет создать более качественный продукт и сократить число багов в релизе. Кроме того, это упрощает дальнейшее обслуживание продукта. Для этого нужно выбрать типы тестов и их количество, а также найти соотношение между модульными, интеграционными и сквозными тестами из-за ограниченности ресурсов. Анализируя разные параметры, сильные и слабые стороны конкретной разновидности тестов, можно принимать более рациональные решения.

Важно, что тестирование увеличивает время разработки. Чем больше тестов мы хотим провести, тем больше времени мы потратим. Иногда трудно реализовать хорошие сквозные тесты, если не планировать их в пределах заданного срока. Таким образом, некоторые виды тестов необходимо планировать так же, как добавление нового функционала в программу, — заранее, а не задним числом.

1.2. ПРОГРАММНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ И ПОЧЕМУ ОНИ РАБОТАЮТ НЕ ВСЕГДА

Паттерны проектирования — Строитель (Builder), Декоратор (Decorator), Прототип (Prototype) и многие другие — появились много лет назад. Они предоставляют проверенные на опыте решения многих известных задач. Я настоятельно рекомендую изучить эти паттерны (см. книгу «Design Patterns: Elements of Reusable Object-Oriented Software»[1]) и использовать их в коде, чтобы сделать

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

его более простым в обслуживании и масштабировании — да и просто более качественным. С другой стороны, их следует применять с осторожностью, потому что реализация этих паттернов сильно зависит от контекста. Как вы уже поняли, я пытаюсь показать, что каждое решение в программном продукте подразумевает компромиссы и имеет последствия.

Чтобы понять компромиссы на уровне программного кода, я продемонстрирую паттерн Одиночка (https://refactoring.guru/design-patterns/singleton). Он был введен как механизм совместного использования состояния всеми компонентами. Одиночка — единственный экземпляр, существующий на протяжении всего срока жизни приложения. К нему обращаются все остальные классы. Допустим, вам нужно создать приватный конструктор, чтобы предотвратить создание нового экземпляра. Реализовать паттерн Одиночка для этого несложно, как показывает следующий листинг.

Листинг 1.3. Реализация паттерна Одиночка

```
public class Singleton {
  private static Singleton instance;
  private Singleton() {}

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Есть только один способ получить экземпляр Одиночки: через метод `getInstance()`, возвращающий единственный экземпляр, который может безопасно использоваться разными компонентами. Предполагается, что каждый раз, когда коду на стороне вызова нужно обратиться к одиночному экземпляру, он делает это через `getInstance()`. Далее мы рассмотрим другой сценарий использования, который не требует применения этого метода каждый раз при обращении к экземпляру. На первый взгляд паттерн позволяет быстро добиться успеха; вы получаете возможность совместно использовать код через глобальный одиночный экземпляр. Казалось бы, в чем компромисс?

Рассмотрим этот паттерн в другом контексте. Что произойдет, если применить его в многопоточной среде? Если сразу несколько потоков вызовут `getInstance()` одновременно, может возникнуть состояние гонки (race condition). В этой ситуации код создаст два экземпляра Одиночки. Разумеется, это нарушит инварианты паттерна и может привести к сбою системы. Чтобы этого избежать, необходимо добавить синхронизацию перед выполнением логики инициализации, как показано в следующем листинге.

Листинг 1.4. Синхронизация для потоково-безопасного паттерна Одиночка

```
public class SystemComponentSingletonSynchronized {
  private static SystemComponent instance;

  private SystemComponentSingletonSynchronized() {}

  public static synchronized SystemComponent getInstance() {
    if (instance == null) {
      instance = new SystemComponent();
    }

    return instance;
  }
}
```

Начинает блок синхронизации

Блок `synchronized` предотвращает обращение к этой логике из двух потоков. Все потоки, кроме одного, блокируются и ожидают логики инициализации. На первый взгляд все работает, как ожидалось. Но Одиночка в многопоточном сценарии может значительно снизить производительность кода, что важно учитывать, если она первоочередная.

Инициализация — первый процесс, в ходе которого несколько потоков должны блокироваться и ожидать. А после создания одиночного экземпляра все обращения к нему должны синхронизироваться. Одиночка может вызвать конкуренцию потоков (http://mng.bz/M2nn), что создаст серьезные риски для производительности. Это происходит, когда имеется общий экземпляр объекта, к которому одновременно обращаются несколько потоков.

Синхронизированный метод `getInstance()` позволяет только одному потоку войти в критическую секцию кода, тогда как остальные вынуждены ожидать снятия блокировки. Когда один поток выходит из критической секции, следующий в очереди может обратиться к ней. Недостаток такого подхода в том, что он создает необходимость синхронизации и может значительно замедлить программу. В двух словах, каждый раз, когда в коде выполняется синхронизируемый вызов, это может привести к дополнительным затратам.

Из примера можно сделать вывод, что существует компромисс между производительностью кода при использовании Одиночки в однопоточном или многопоточном контексте. Но для нас важен контекст, в котором код выполняется. Если он работает без параллельного выполнения или одиночный экземпляр не используется совместно разными потоками, компромисс не проявляется. Но при совместном использовании Одиночки необходимо обеспечить его потоковую безопасность, что потенциально влияет на производительность. Знание этого компромисса позволит принимать рациональные решения, касающиеся архитектуры и кода.

Если окажется, что у выбранного варианта больше минусов, чем плюсов, решение можно изменить. Так, в примере с Одиночкой ситуацию можно улучшить, применив один из двух паттернов.

В первом варианте используется метод блокировки с двойной проверкой. Этот способ отличается тем, что перед входом в критическую (синхронизированную) секцию необходимо проверить экземпляр на наличие `null`. Если условие выполняется, можно продолжать вход в критическую секцию, в противном случае в этом нет необходимости — нужно просто вернуть существующий одиночный экземпляр. Эта реализация блокировки представлена в следующем листинге.

Листинг 1.5. Блокировка с двойной проверкой для паттерна Одиночка

```
private volatile static SystemComponent instance;

public static SystemComponent getInstance() {
  if (instance == null) {                          <-- Если экземпляр не содержит null,
    synchronized (ThreadSafeSingleton.class) {         не входить в критическую секцию
      if (instance == null) {
        instance = new SystemComponent();
      }
    }
  }
  return instance;
}
```

Применение этого паттерна значительно снижает необходимость в синхронизации и уровень конкуренции между потоками. Эффект синхронизации будет наблюдаться только при запуске, когда каждый поток пытается инициализировать одиночный экземпляр.

Второй паттерн, который можно применить, — привязка к потоку. Он позволяет закрепить состояние за конкретным потоком. Однако при этом необходимо понимать, что паттерн Одиночка перестает существовать на уровне глобального приложения — на каждый поток будет приходиться один экземпляр объекта. Если в приложении существуют *N* потоков, то в нем будут использоваться *N* экземпляров.

При использовании данного паттерна каждый поток в коде владеет экземпляром объекта, видимым для конкретного потока и привязанным к нему. Благодаря этому потоки не конкурируют за доступ к объекту. Объект принадлежит потоку и не используется совместно. В Java желаемого эффекта можно добиться при помощи класса `ThreadLocal` (http://mng.bz/aD8B). Он позволяет обернуть системный компонент, который будет привязан к конкретному потоку. С точки зрения кода объект находится внутри экземпляра `ThreadLocal`, как показывает следующий листинг.

Листинг 1.6. Привязка к потоку с использованием ThreadLocal

```
private static ThreadLocal<SystemComponent> threadLocalValue = new
    ThreadLocal<>();

public static void set() {
  threadLocalValue.set(new SystemComponent());
}

public static void executeAction() {
  SystemComponent systemComponent = threadLocalValue.get();
}

public static SystemComponent get() {
  return threadLocalValue.get();
}
```

Логика привязки `SystemComponent` к конкретному потоку инкапсулируется в экземпляре `ThreadLocal`. Когда поток A вызывает метод `set()`, внутри `ThreadLocal` создается новый экземпляр `SystemComponent`. Важно, что этот экземпляр доступен только для этого потока. Если другой поток (скажем, B) вызовет `executeAction()` без предварительного вызова `set()`, он получит `null`-экземпляр `SystemComponent`, потому что для этого потока компонент еще не был создан вызовом `set()`. Новый экземпляр, предназначенный для этого потока, будет создан и станет доступен только после того, как поток B вызовет метод `set()`.

Происходящее можно упростить, передав поставщик (supplier) при вызове метода `withInitial()`. Он будет вызван, если потоково-локальный объект еще не получил значения, так что риск получить `null` отсутствует. В следующем листинге показана возможная реализация.

Листинг 1.7. Привязка к потоку с исходным значением

```
static ThreadLocal<SystemComponent> threadLocalValue =
    ThreadLocal.withInitial(SystemComponent::new);
```

Применение этого паттерна устраняет конкуренцию, что улучшает производительность. Обратной стороной такого решения становится усложнение системы.

ПРИМЕЧАНИЕ

Каждый раз, когда код на стороне вызова хочет обратиться к одиночному экземпляру, обращение не обязано использовать метод `getInstance()`. Можно обратиться к одиночному экземпляру один раз и присвоить его переменной (ссылке). После этого дальнейшие обращения могут получить доступ к одиночному объекту по ссылке без необходимости вызывать `getInstance()`. Такое решение снижает конкуренцию потоков.

Одиночный экземпляр также можно внедрить в другие компоненты, которые должны его использовать. В идеале приложение создает все компоненты в одном месте и помещает их в сервисы (например, с применением внедрения

зависимостей). В таком случае паттерн Одиночка может вообще не понадобиться. Вы просто создаете один экземпляр объекта, который должен находиться в общем доступе, и внедряете его во все зависимые сервисы (http://mng.bz/g4dE). Альтернатива этому — использование перечисляемого типа, в основе которого лежит паттерн Одиночка. Чтобы проверить предположения, измерим временные характеристики кода.

1.2.1. Измерение скорости выполнения

К этому моменту мы создали три потоково-безопасные реализации паттерна Одиночка:

- с синхронизацией для всех операций;
- с блокировкой с двойной проверкой;
- с привязкой к потокам (с использованием `ThreadLocal`).

Предполагается, что первая версия будет самой медленной, но данных об этом у нас пока нет. Создадим тест производительности, который будет проверять все три реализации. Воспользуемся инструментарием проверки производительности JMH (https://openjdk.java.net/projects/code-tools/jmh/), который мы еще применим в дальнейшем для проверки быстродействия кода.

Создадим хронометражный тест, который выполняет 50 000 операций получения (одиночного) объекта `SystemComponent` (листинг 1.8). В нем будут реализованы три теста, использующих разные подходы к функциональности паттерна Одиночка. Чтобы проверить, как конкуренция потоков влияет на производительность, код будет выполняться в 100 одновременно работающих потоках. Результаты (среднее время выполнения) будут выведены в миллисекундах.

Листинг 1.8. Создание хронометражных тестов реализаций паттерна Одиночка

```
@Fork(1)
@Warmup(iterations = 1)
@Measurement(iterations = 1)
@BenchmarkMode(Mode.AverageTime)
@Threads(100)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class BenchmarkSingletonVsThreadLocal {
  private static final int NUMBER_OF_ITERATIONS = 50_000;

  @Benchmark
  public void singletonWithSynchronization(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
      blackhole.consume(
➥ SystemComponentSingletonSynchronized.getInstance());
    }
  }
```

Код выполняется в 100 одновременно работающих потоках

В первом тесте используется SystemComponent SingletonSynchronized

```
  @Benchmark
  public void singletonWithDoubleCheckedLocking(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
      blackhole.consume(
➥ SystemComponentSingletonDoubleCheckedLocking.getInstance());
    }
  }

  @Benchmark
  public void singletonWithThreadLocal(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
      blackhole.consume(SystemComponentThreadLocal.get());
    }
  }
}
```

Тесты для System ComponentSingleton DoubleCheckedLocking

Получить результаты для SystemComponentThreadLoc

Выполнив тест, мы получаем среднее время на 50 000 вызовов для 100 одновременных потоков. В конкретной среде числа могут быть другими, но общие тенденции останутся теми же, как показывает следующий листинг.

Листинг 1.9. Просмотр результатов хронометражного теста реализаций паттерна Одиночка

```
Benchmark                                                          Mode  Cnt    Score   Error  Units
CH01.BenchmarkSingletonVsThreadLocal.singletonWithDoubleCheckedLocking  avgt          2.629           ms/op
CH01.BenchmarkSingletonVsThreadLocal.singletonWithSynchronization       avgt        316.619           ms/op
CH01.BenchmarkSingletonVsThreadLocal.singletonWithThreadLocal           avgt          5.622           ms/op
```

Из результатов видно, что реализация `singletonWithSynchronization` действительно оказалась самой медленной. Среднее время выполнения логики хронометража составило около 300 мс (миллисекунд). Затем идут два решения с более высокими результатами. Реализация `singletonWithDoubleCheckedLocking` показывает лучший результат (около 2,6 мс), а реализация `singletonWithThreadLocal` завершилась за 5,6 мс. Можно сделать вывод, что усовершенствование исходной версии паттерна Одиночка обеспечивает примерно 50-кратное повышение производительности для потоково-локального решения и 115-кратное для решения с блокировкой с двойной проверкой.

Проверяя предположения, можно принимать эффективные решения в многопоточном контексте. Если потребуется выбрать одно решение вместо другого при сравнимой производительности, можно отдать предпочтение более прямолинейному решению. Тем не менее без реальных данных трудно сделать полностью рациональный выбор.

Теперь рассмотрим компромиссы для архитектурных решений. В следующем разделе вы узнаете о микросервисных и монолитных архитектурах и о связанных с ними компромиссах.