Чтобы начать писать хорошие автотесты, нужно ﻿разобраться, в чем плюсы и минусы ручного и автоматизированного тестирования.

| **Ручные тесты** | **Автотесты** |
| --- | --- |
| Шаги могут быть описаны достаточно абстрактно, поэтому тестировщик может увидеть новые проблемы, следуя сценарию теста.  В ручном тесте может содержаться шаг "Зарегистрировать нового пользователя" без пояснений, как это делать, в автотесте должны быть строго описаны данные для теста и сценарий регистрации. | Строгий порядок шагов и их чёткая детализация. Проверяют только то, что было заложено изначально в сценарий теста, поэтому могут быть пропущены новые ошибки в продукте. |
| - Из-за абстрактности описания шагов ручные тесты иногда могут приводить к ложноположительным результатам, когда ожидаемый результат достигнут, но в реальности тестировщик выполнил сценарий неправильно, пропустив баг. | + Запуск автотестов обычно приводит к одним и тем же результатам, т.к. сценарий описан точно. |
| - Требуется много времени для проверки регрессии, что приводит к усталости тестировщика и ошибкам в проверках из-за человеческого фактора (человеку сложно долго выполнять монотонную работу). | + Можно запускать на каждый коммит в тестируемом приложении, что позволяет раньше обнаруживать ошибки в продукте. |

Отдельно стоит описать отрицательные стороны автоматизированного тестирования:

* возможная нестабильность теста, которая не связана с качеством самого теста, а вызывается внешними проблемами: нестабильное сетевое соединение, проблемы с серверами, обновление кода продукта в момент запуска тестов;
* требуется достаточно много времени на разработку и поддержку набора автотестов.

Несмотря на наличие минусов автотестов, в большинстве случаев их использование на проекте помогает быстрее находить ошибки в коде приложения и поддерживать качество продукта на достаточном уровне. Автотесты помогают отделу тестирования оптимизировать свою работу, чтобы сделать счастливее пользователей, которые быстрее получают новые фичи и меньше страдают от ошибок в продукте, так как разработчики раньше узнают о багах и могут заранее принять меры для их устранения, не откладывая релиз продукта. Таким образом, благодаря автотестам налаживается бизнес, который может увеличить скорость внедрения фич, приносящих прибыль.

Ручные тесты в идеальном случае остаются только на этапе проверки новых фич и в виде исследовательского тестирования, которое позволяет найти проблемы в сложных пользовательских сценариях.

**Юнит-тесты и интеграционные тесты**

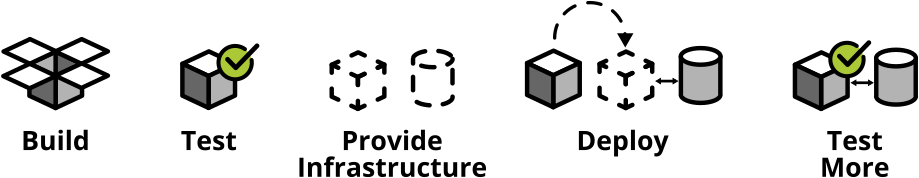
Если вы работаете в тестировании, то уже знаете разницу между юнит-тестами и интеграционными тестами. Юнит-тесты проверяют очень маленький кусок кода, обычно конкретную функцию, и чаще всего их пишут разработчики, которые хорошо понимают возможные крайние случаи для своего стека технологий. Интеграционные тесты проверяют взаимодействие сразу нескольких систем. Они могут создаваться и поддерживаться как разработчиками и тестировщиками, так и аналитиками (если для них разработан удобный фреймворк для написания тестов).

Юнит-тесты всегда автоматизированы, так как проверяют непосредственно работу кода. Интеграционные тесты могут быть ручными и автоматизированными. Иногда выделяют отдельную категорию end-to-end (е2е) тестов, которые проверяют полный стек технологий приложения и пользовательский сценарий взаимодействия с приложением как с черным ящиком. Если говорить про UI-тесты, которые разрабатываются с помощью Selenium, то их стоит отнести к разряду end-to-end тестов, так как они проверяют совместную работу всех систем web-продукта: работу frontend и backend, работу базы данных, дополнительные сервисы, такие как аналитика, платежные системы и так далее.

**Пирамида тестов»** — метафора, которая означает группировку тестов программного обеспечения по разным уровням детализации. Она также даёт представление, сколько тестов должно быть в каждой из этих групп. Несмотря на то, что концепция тестовой пирамиды существует довольно давно, многие команды разработчиков по-прежнему пытаются неправильно реализовать её на практике должным образом. В этой статье рассматривается первоначальная концепция тестовой пирамиды и показано, как её воплотить в жизнь. Она показывает, какие виды тестов следует искать на разных уровнях пирамиды, и даёт практические примеры, как их можно реализовать.

Перед выпуском программное обеспечение нужно тестировать. По мере взросления софтверной отрасли созрели и подходы к тестированию. Вместо мириадов живых тестировщиков разработчики перешли к автоматизации большей части тестов. Автоматизация тестов позволяет узнать о баге в считанные секунды и минуты после его внесения в код, а не через несколько дней или недель.  
  
Резко сокращённый цикл обратной связи, подпитываемый автоматизированными тестами, идёт рука об руку с гибкими практиками разработки, непрерывной доставкой и культурой DevOps. Эффективный подход к тестированию обеспечивает быструю и уверенную разработку.  
  
В этой статье рассматривается, как должен выглядеть хорошо сформированный набор тестов, чтобы быть гибким, надёжным и поддерживаемым — независимо от того, создаете ли вы архитектуру микросервисов, мобильные приложения или экосистемы IoT. Мы также детально рассмотрим создание эффективных и удобочитаемых автоматизированных тестов.

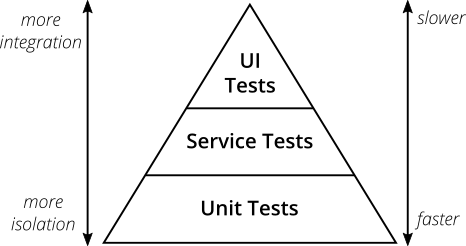
# **Важность автоматизации (тестов)**

Программное обеспечение стало неотъемлемой частью мира, в котором мы живём. Оно переросло первоначальную единственную цель увеличить эффективность бизнеса. Сегодня каждая компания стремится стать первоклассной цифровой компанией. Все мы каждый день выступаем пользователями всё большего количества ПО. Скорость инноваций возрастает.  
  
Если хотите идти в ногу со временем, нужно искать более быстрые способы доставки ПО, не жертвуя его качеством. В это может помочь непрерывная доставка — это практика, которая автоматически гарантирует, что ПО может быть выпущено в продакшн в любое время. При непрерывной доставке используется конвейер сборки для автоматического тестирования ПО и его развёртывания в тестовой и рабочей средах.  
  
Вскоре сборка, тестирование и развёртывание постоянно растущего количества ПО вручную становится невозможной — если только вы не хотите тратить всё своё время на выполнение вручную рутинных задач вместо доставки рабочего софта. Единственный путь — автоматизировать всё, от сборки до тестирования, развёртывания и инфраструктуры.  
  
  
  
*Рис. 1. Использование конвейеров сборки для автоматического и надёжного ввода ПО в эксплуатацию*

Традиционно тестирование требовало чрезмерной ручной работы через развертывание в тестовой среде, а затем тестов в стиле чёрного ящика, например, кликанием повсюду в пользовательском интерфейсе с наблюдением, появляются ли баги. Часто эти тесты задаются тестовыми сценариями, чтобы гарантировать, что тестировщики всё последовательно проверят.  
  
Очевидно, что тестирование всех изменений вручную занимает много времени, оно однообразное и утомительное. Однообразие скучно, а скука приводит к ошибкам.  
  
К счастью, есть прекрасный инструмент для однообразных задач: *автоматизация*.  
  
Автоматизация однообразных тестов изменит вашу жизнь как разработчика. Автоматизируйте тесты, и вам больше не придётся бездумно следовать клик-протоколам, проверяя корректность работы программы. Автоматизируйте тесты, и вы не моргнув глазом измените кодовую базу. Если вы когда-либо пробовали делать крупномасштабный рефакторинг без надлежащего набора тестов, я уверен, вы знаете, в какой ужас это может превратиться. Как вы узнаете, если случайно сделаете ошибку в процессе? Ну, придётся щёлкать вручную по всем тестовым случаям, как же ещё. Но будем честными: вам это действительно нравится? Как насчёт того, чтобы даже после крупномасштабных изменений любые баги проявляли себя в течение нескольких секунд, пока вы пьёте кофе? По-моему, это гораздо приятнее.

# **Пирамида тестов**

Если серьёзно подходить к автоматическим тестам, то есть одна ключевая концепция: **пирамида тестов**. Её представил Майк Кон в своей книге *«Scrum: гибкая разработка ПО»* (Succeeding With Agile. Software Development Using Scrum). Это отличная визуальная метафора, наталкивающая на мысль о разных уровнях тестов. Она также показывает объём тестов на каждом уровне.



*Рис. 2. Пирамида тестов*  
  
Оригинальная пирамида тестов Майка Кона состоит из трёх уровней (снизу вверх):

1. Юнит-тесты.
2. Сервисные тесты.
3. Тесты пользовательского интерфейса.

К сожалению, при более тщательном концепция кажется недостаточной. Некоторые утверждают, что либо именования, либо некоторые концептуальные аспекты пирамиды тестов Майка Кона не идеальны, и я должен согласиться. С современной точки зрения пирамида тестов кажется чрезмерно упрощённой и поэтому может вводить в заблуждение.  
  
Тем не менее, из-за своей простоты суть тестовой пирамиды представляет хорошее эмпирическое правило, когда дело доходит до создания собственного набора тестов. Из этой пирамиды главное запомнить два принципа:

1. Писать тесты разной детализации.
2. Чем выше уровень, тем меньше тестов.

Придерживайтесь формы пирамиды, чтобы придумать здоровый, быстрый и поддерживаемый набор тестов. Напишите *много* маленьких и быстрых *юнит-тестов*. Напишите *несколько* более общих тестов и *совсем мало* высокоуровневых сквозных тестов, которые проверяют приложение от начала до конца. Следите, что у вас в итоге не получился [тестовый рожок мороженого](https://watirmelon.blog/testing-pyramids/), который станет кошмаром в поддержке и будет слишком долго выполняться.  
  
Не привязывайтесь слишком сильно к названиям отдельных уровней пирамиды тестов. На самом деле они могут ввести в заблуждение: термин «сервисный тест» трудно понять (сам Кон заметил, что [многие разработчики полностью игнорируют этот уровень](https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid)). В наше время фреймворков для одностраничных приложений вроде React, Angular, Ember.js и других становится очевидным, что тестам UI не место на вершине пирамиды — вы прекрасно можете протестировать UI во всех этих фреймворках.  
  
Учитывая недостатки оригинальных названий в пирамиде, вполне нормально придумать другие имена для своих уровней тестов. Главное, чтобы они соответствовали вашему коду и терминологии, принятой в вашей команде.

# **Пример приложения**

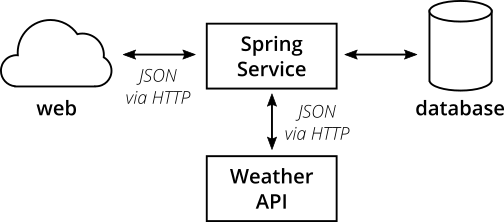
Я написал [простой микросервис](https://github.com/hamvocke/spring-testing) с тестами из разных уровней пирамиды.  
  
Это пример типичного микросервиса. Он предоставляет интерфейс REST, общается с БД и извлекает информацию из стороннего сервиса REST. Он реализован на [Spring Boot](https://projects.spring.io/spring-boot/) и должен быть понятен даже если вы никогда не работали со Spring Boot.  
  
Обязательно проверьте [код на Github](https://github.com/hamvocke/spring-testing). В файле readme инструкции для запуска приложения и автоматических тестов на вашем компьютере.

### **Функциональность**

У приложения простая функциональность. Оно обеспечивает интерфейс REST с тремя конечными точками:  
  
GET /hello  
Возвращает *"Hello World"*. Всегда.  
  
GET /hello /{lastname}  
Ищет человека с указанной фамилией. Если человек известен, возвращает *"Hello {Firstname} {Lastname}"*.  
  
GET /weather  
Возвращает текущие погодные условия в *Гамбурге, Германия*.

### **Высокоуровневые структуры**

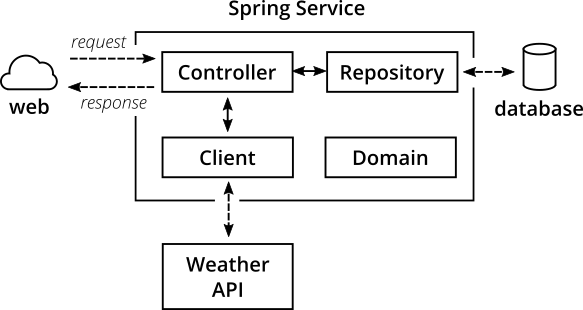
На высоком уровне у системы следующая структура:



*Рис. 3. Высокоуровневая структура микросервиса*  
  
Наш микросервис обеспечивает интерфейс REST по HTTP. Для некоторых конечных точек сервис получает информацию из БД. В других случаях обращается по HTTP к внешнему [API](https://darksky.net/) для получения и отображения текущей погоды.

### **Внутренняя архитектура**

Изнутри у Spring Service типичная архитектура для Spring:



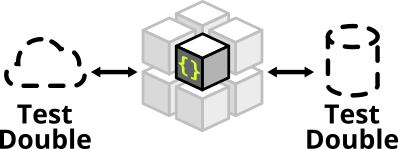
*Рис. 4. Внутренняя структура микросервиса*

* Классы Controller предоставляют конечные точки *REST*, обрабатывают запросы *HTTP* и ответы.
* Классы Repository взаимодействуют с *базой данных*, отвечают за запись и чтение данных в/из постоянного хранилища.
* Классы Client взаимодействуют с другими API, в нашем случае — забирают данные *JSON* по *HTTPS* с погодного API на darksky.net.
* Классы Domain захватывают [модель домена](https://en.wikipedia.org/wiki/Domain_model), включая логику домена (которая, честно говоря, в нашем случае довольно тривиальна).

Опытные разработчики Spring могут заметить, что здесь отсутствует часто используемый слой: многие вдохновлённые [проблемно-ориентированным проектированием](https://en.wikipedia.org/wiki/Domain-driven_design) разработчики создают *слой сервисов*, состоящий из классов *сервисов*. Я решил не включать его в приложение. Одна из причин в том, что наше приложение достаточно простое, а слой сервисов станет ненужным уровнем косвенности. Другая причина в том, что на мой взгляд люди часто переусердствуют с этими слоями. Нередко приходится видеть кодовые базы, где классы сервисов охватывают всю бизнес-логику. Модель домена становится просто слоем для данных, а не для поведения ([анемичная модель домена](https://en.wikipedia.org/wiki/Anemic_domain_model)). Для каждого нетривиального приложения так теряются большие возможности для хорошей структуризации кода и тестируемости, а также не в полной мере используется мощь объектной ориентации.  
  
Наши репозитории просты и обеспечивают простую функциональность CRUD. Для простоты кода я использовал [Spring Data](http://projects.spring.io/spring-data/). Он даёт простую и универсальную реализацию репозитория CRUD, а также заботится о том, чтобы развернуть для наших тестов БД в памяти, а не использовать реальную PostgreSQL, как это было бы в продакшне.  
  
Взгляните на кодовую базу и познакомьтесь с внутренней структурой. Это полезно для следующего шага: тестирования приложения!

# **Юнит-тесты**

Основа вашего набора тестов состоит из юнит-тестов (модульных тестов). Они проверяют, что отдельный юнит (*тестируемый субъект*) кодовой базы работает должным образом. Модульные тесты имеют максимально узкую область среди всех тестов в наборе тестов. Количество юнит-тестов в наборе значительно превышает количество любых других тестов.



*Рис. 5. Обычно юнит-тест заменяет внешних пользователей тестовыми дублями*

### **Что такое юнит?**

Если вы спросите трёх разных людей, что означает *«юнит»* в контексте юнит-тестов, то вероятно получите четыре разных, слегка отличающихся ответа. В определённой степени это вопрос вашего собственного определения — и это нормально, что здесь нет общепринятого канонического ответа.  
  
Если вы пишете на функциональном языке, то юнитом скорее всего будет отдельная функция. Ваши юнит-тесты вызовут функцию с различными параметрами и обеспечат возврат ожидаемых значений. В объектно-ориентированном языке юнит может варьироваться от отдельного метода до целого класса.

### **Общительные и одинокие тесты**

Некоторые утверждают, что всех участников (например, вызываемые классы) тестируемого субъекта следует заменить на *имитации* (mocks) или *заглушки* (stubs), чтобы создать идеальную изоляцию, избежать побочных эффектов и сложной настройки теста. Другие утверждают, что на имитации и заглушки следует заменять только участников, которые замедляют тест или проявляют сильные побочные эффекты (например, классы с доступом к БД или сетевыми вызовами).  
  
[Иногда](https://martinfowler.com/bliki/UnitTest.html) эти два вида юнит-тестов называют **одинокими** (solitary) в случае тотального применения имитаций и заглушек или **общительными** (sociable) в случае реальных коммуникаций с другими участниками (эти термины придумал Джей Филдс для книги [«Эффективная работа с юнит-тестами»](https://leanpub.com/wewut)). Если у вас есть немного свободного времени, можете спуститься в кроличью нору и [разобраться в преимуществах и недостатках](https://martinfowler.com/articles/mocksArentStubs.html) разных точек зрения.  
  
Но в итоге не имеет значения, какой тип тестов вы выберете. Что реально важно, так это их автоматизация. Лично я постоянно использую оба подхода. Если неудобно работать с реальными участниками, я буду обильно использовать имитации и заглушки. Если чувствую, что привлечение реального участника даёт больше уверенности в тесте, то заглушу только самые дальние части сервиса.

### **Имитации и заглушки**

Имитации (mocks) и заглушки (stubs) — это два разных типа [тестовых дублёров](https://martinfowler.com/bliki/TestDouble.html) (вообще их больше). Многие используют термины взаимозаменяемо. Думаю, что лучше соблюдать точность и держать в уме конкретные свойства каждого из них. К объектам из продакшна тестовые дублёры создают реализацию для тестов.  
  
Проще говоря, вы заменяете реальную вещь (например, класс, модуль или функцию) поддельной копией. Подделка выглядит и действует как оригинал (даёт такие же ответы на те же вызовы методов), но это заранее установленные ответы, которые вы сами определяете для юнит-теста.  
  
Тестовые дублёры используются не только в юнит-тестах. Более сложные дублёры применяются для контролируемой имитации целых частей вашей системы. Однако в юнит-тестах используется особенно много имитаций и заглушек (в зависимости от того, предпочитаете вы общительные или одиночные тесты) просто потому что множество современных языков и библиотек позволяют легко и удобно их создавать.  
  
Независимо от выбранной технологии, в стандартной библиотеке вашего языка или какой-то популярной сторонней библиотеке уже есть элегантный способ настройки имитаций. И даже для написания собственных имитаций с нуля достаточно всего лишь написать поддельный класс/модуль/функцию с той же подписью, что и реальная, и настройки имитации для теста.  
  
Ваши юнит-тесты будут работать очень быстро. На приличной машине можно прогнать тысячи модульных тестов за нескольких минут. Тестируйте изолированно небольшие фрагменты кодовой базы и избегайте контактов с БД, файловой системой и HTTP-запросов (ставя здесь имитации и заглушки), чтобы сохранить высокую скорость.  
  
Поняв основы, со временем вы начнёте всё более свободно и легко писать юнит-тесты. Заглушка внешних участников, настройка входных данных, вызов тестируемого субъекта — и проверка, что возвращаемое значение соответствует ожидаемому. Посмотрите на [разработку через тестирование](https://en.wikipedia.org/wiki/Test-driven_development) (TDD), и пусть юнит-тесты направляют вашу разработку; если они применяются правильно, это поможет попасть в мощный поток и создать хорошую поддерживаемую архитектуру, автоматически выдавая всеобъемлющий и полностью автоматизированный набор тестов. Но это не универсальное решение. Попробуйте и посмотрите сами, подходит ли TDD в вашем конкретном случае.

### **Что тестировать?**

Хорошо, что юнит-тесты можно писать для всех классов кода продакшна, независимо от их функциональности или того, к какому уровню внутренней структуры они принадлежат. Юнит-тесты подходят для контроллеров, репозиториев, классов предметной области или программ считывания файлов. Просто придерживайтесь практического правила **один тестовый класс на один класс продакшна**.  
  
Юнит-тест должен как минимум **протестировать *открытый* интерфейс класса**. закрытые методы всё равно нельзя протестировать, потому что их нельзя вызвать из другого тестового класса. *Защищённые* или *доступные* лишь в пределах пакета (package-private) методы доступны из тестового класса (учитывая, что структура пакета тестового класса такая же, как на продакшне), но тестирование этих методов может уже зайти слишком далеко.  
  
Когда дело доходит до написания юнит-тестов, есть тонкая черта: они должны гарантировать, что проверены все нетривиальные пути кода, включая дефолтный сценарий и пограничные ситуации. В то же время они не должны быть слишком тесно привязаны к реализации.  
  
Почему так?  
  
Тесты, слишком привязанные к коду продакшна, быстро начинают раздражать. Как только вы осуществляете рефакторинг кода (то есть изменяете внутреннюю структуру кода без изменения внешнего поведения), модульные тесты сразу ломаются.  
  
Таким образом, вы теряете важное преимущество юнит-тестов: действовать в качестве системы безопасности для изменений кода. Вы скорее устанете от этих глупых тестов, которые падают каждый раз после рефакторинга, принося больше проблем, чем пользы; чья вообще была эта дурацкая идея внедрить тесты?  
  
Чем же делать? Не отражайте в модульных тестах внутреннюю структуру кода. Тестируйте наблюдаемое поведение. Например:  
  
*если я введу значения x и y, будет ли результат z?*  
  
вместо этого:  
  
*если я введу x и y, то обратится ли метод сначала к классу А, затем к классу Б, а затем сложит результаты от класса А и класса Б?*  
  
Как правило, закрытые методы следует рассматривать как деталь реализации. Вот почему даже не должно появляться желание их проверить.  
  
Часто я слышу от противников модульного тестирования (или TDD), что написание юнит-тестов становится бессмысленным, если нужно проверить все методы для большого охвата тестирования. Они часто ссылаются на сценарии, где чрезмерно нетерпеливый тимлид заставил писать модульные тесты для геттеров и сеттеров и прочего тривиального кода, чтобы выйти на 100% тестового покрытия.  
  
Это совершенно неправильно.  
  
Да, вы должны *протестировать публичный интерфейс*. Но ещё более важно *не тестировать тривиальный код*. Не волнуйтесь, [Кент Бек это одобряет](https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/). Вы ничего не получите от тестирования простых геттеров или сеттеров или других тривиальных реализаций (например, без какой-либо условной логики). И вы сэкономите время, так что сможете посидеть ещё на одном совещании, ура!

**Но мне *очень* нужно проверить этот закрытый метод**  
Если вы когда-нибудь окажетесь в ситуации, когда вам *очень-очень* нужно проверить закрытый метод, нужно сделать шаг назад и спросить себя: почему?  
  
Уверен, что здесь скорее проблема дизайна. Скорее всего, вы чувствуете необходимость протестировать закрытый метод, потому что он сложный, а тестирование метода через открытый интерфейс класса требует слишком неудобной настройки.  
  
Всякий раз, когда я оказываюсь в такой ситуации, я обычно прихожу к выводу, что тестируемый класс переусложнён. Он делает слишком много и нарушает принцип единой ответственности — один из пяти принципов [SOLID](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)).  
  
Для меня часто работает решение разделить исходный класс на два класса. Часто после минуты-другой размышлений находится хороший способ разбить большой класс на два меньших с индивидуальной ответственностью. Я перемещаю закрытый метод (который срочно надо протестировать) в новый класс и позволяю старому классу вызвать новый метод. Вуаля, неудобный для тестирования закрытый метод теперь публичен и легко тестируется. Кроме того, я улучшил структуру кода, внедрив принцип единой ответственности.

### **Cтруктура теста**

Хорошая структура всех ваших тестов (не только модульных) такова:

1. Настройка тестовых данных.
2. Вызов тестируемого метода.
3. Проверка, что возвращаются ожидаемые результаты.

Есть хорошая мнемоника для запоминания этой структуры: три A ([Arrange, Act, Assert](https://xp123.com/articles/3a-arrange-act-assert/)). Можно использовать и другую мнемонику с корнями в BDD (разработка, основанная на описании поведения). Это триада [дано, когда, тогда](https://martinfowler.com/bliki/GivenWhenThen.html), где «дано» отражает настройку, «когда» — вызов метода, а «тогда» — утверждение.  
  
Этот шаблон можно применить и к другим, более высокоуровневым тестам. В каждом случае они гарантируют, что тесты остаются лёгкими и читаемыми. Кроме того, написанные с учётом этой структуры тесты обычно короче и выразительнее.

### **Реализация юнит-теста**

Теперь мы знаем, что именно тестировать и как структурировать юнит-тесты. Пришло время посмотреть на реальный пример.  
  
Возьмем упрощённую версию класса ExampleController.

@RestController

**public** **class** **ExampleController** {

**private** **final** PersonRepository personRepo;

@Autowired

**public** **ExampleController**(**final** PersonRepository personRepo) {

**this**.personRepo = personRepo;

}

@GetMapping("/hello/{lastName}")

**public** String **hello**(@PathVariable **final** String lastName) {

Optional<Person> foundPerson = personRepo.findByLastName(lastName);

**return** foundPerson

.map(person -> String.format("Hello %s %s!",

person.getFirstName(),

person.getLastName()))

.orElse(String.format("Who is this '%s' you're talking about?",

lastName));

}

}

Юнит-тест для метода hello(lastname) может выглядеть таким образом:

**public** **class** **ExampleControllerTest** {

**private** ExampleController subject;

@Mock

**private** PersonRepository personRepo;

@Before

**public** **void** **setUp**() **throws** Exception {

initMocks(**this**);

subject = **new** ExampleController(personRepo);

}

@Test

**public** **void** **shouldReturnFullNameOfAPerson**() **throws** Exception {

Person peter = **new** Person("Peter", "Pan");

given(personRepo.findByLastName("Pan"))

.willReturn(Optional.of(peter));

String greeting = subject.hello("Pan");

assertThat(greeting, is("Hello Peter Pan!"));

}

@Test

**public** **void** **shouldTellIfPersonIsUnknown**() **throws** Exception {

given(personRepo.findByLastName(anyString()))

.willReturn(Optional.empty());

String greeting = subject.hello("Pan");

assertThat(greeting, is("Who is this 'Pan' you're talking about?"));

}

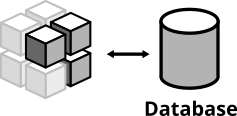
}

Мы пишем юнит-тесты в [JUnit](http://junit.org/), стандартном фреймворке тестирования Java. Используем [Mockito](http://site.mockito.org/) для замены реального класса PersonRepository на класс с заглушкой для теста. Эта заглушка позволяет указать предустановленные ответы, которые вернёт метод-заглушка. Подобный подход делает тест более простым и предсказуемым, позволяя легко настроить проверку данных.  
  
Следуя структуре «трёх А» пишем два юнит-теста для положительного и отрицательного случаев, когда искомое лицо не может быть найдено. Положительный тестовый случай создаёт новый объект person и сообщает имитации репозитория возвращать этот объект, когда параметр lastName вызывается со значением *Pan*. Затем тест вызывает тестируемый метод. Наконец, он сравнивает ответ с ожидаемым.  
  
Второй тест работает аналогично, но тестирует сценарий, в котором тестируемый метод не находит объект person для данного параметра.

**Специализированные тестовые хелперы**  
Замечательно, что вы можете писать юнит-тесты для всей кодовой базы независимо от уровня архитектуры вашего приложения. Пример ниже показывает простой юнит-тест для контроллера. К сожалению, когда дело доходит до контроллеров Spring, у этого подхода есть недостаток: контроллер Spring MVC интенсивно использует аннотации с объявлениями прослушиваемых путей, используемых команд HTTP, параметров парсинга URL, параметров запросов и так далее. Простой вызов метода контроллера в юнит-тесте не проверит все эти важные вещи. К счастью, сообщество Spring придумало хороший тестовый хелпер, который можно использовать для улучшенного тестирования контроллера. Обязательно посмотрите [MockMVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-server). Это даст отличный DSL для генерации поддельных запросов к контроллеру и проверки, что всё работает отлично. Я включил [пример](https://github.com/hamvocke/spring-testing/blob/master/src/test/java/example/ExampleControllerAPITest.java) в код. Во многих фреймворках есть тестовые хелперы для упрощения тестов конкретных частей кода. Ознакомьтесь с документацией по своему фреймворку и посмотрите, предлагает ли там какие-либо полезные хелперы для ваших автоматизированных тестов.

# **Интеграционные тесты**

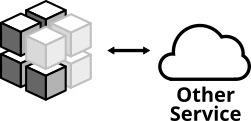
Все нетривиальные приложения интегрированы с некоторыми другими частями (базы данных, файловые системы, сетевые вызовы к другим приложениям). В юнит-тестах вы обычно имитируете их для лучшей изоляции и повышения скорости. Тем не менее, ваше приложение будет реально взаимодействовать с другими частями — и это следует протестировать. Для этого предназначены [интеграционные тесты](https://martinfowler.com/bliki/IntegrationTest.html). Они проверяют интеграцию приложения со всеми компонентами вне приложения.  
  
Для автоматизированных тестов это означает, что нужно запустить не только собственное приложение, но и интегрируемый компонент. Если вы тестируете интеграцию с БД, то при выполнении тестов надо запустить БД. Чтобы проверить чтение файлов с диска нужно сохранить файл на диск и загрузить его в интеграционный тест.  
  
Я ранее упоминал, что юнит-тесты — неопределённый термин. Ещё в большей степени это относится к интеграционным тестам. Для кого-то «интеграция» означает тестирование всего стека вашего приложения в комплексе с другими. Мне нравится более узкое определение и тестирование каждой точки интеграции по отдельности, заменяя остальные сервисы и базы данных тестовыми дублёрами. Вместе с контрактным тестированием и выполнением контрактных тестов на дублёрах и реальных реализациях можно придумать интеграционные тесты, которые быстрее, более независимы и обычно проще в понимании.  
  
Узкие интеграционные тесты живут на границе вашего сервиса. Концептуально они всегда запускают действие, которое приводит к интеграции с внешней частью (файловой системой, базой данных, отдельным сервисом). Тест интеграции БД выглядит следующим образом:



*Рис. 6. Тест на интеграцию БД интегрирует ваш код с реальной базой данных*

1. Запуск базы данных.
2. Подключение приложения к БД.
3. Запуск функции в коде, которая записывает данные в БД.
4. Проверка, что ожидаемые данные записаны в базу путём их чтения из БД.

Другой пример. Тест интеграции вашего сервиса с отдельной службой через REST API может выглядеть следующим образом:



*Рис. 7. Этот вид интеграционного теста проверяет, что приложение способно правильно взаимодействовать с отдельными службами*

1. Запуск приложения.
2. Запуск инстанса отдельной службы (или тестового дублёра с тем же интерфейсом).
3. Запуск функции в коде, которая считывает данные из API внешней службы.
4. Проверка, что приложение правильно разбирает ответ.

Как и модульные тесты, ваши интеграционные тесты можно делать вполне прозрачно (whitebox). Некоторые фреймворки позволяют одновременно запустить и ваше приложение, и имитации отдельных его частей для проверки правильного взаимодействия.  
  
Напишите интеграционные тесты для всех фрагментов кода, где выполняется *сериализация* или *десериализация* данных. Это происходит чаще, чем вы думаете. Подумайте о следующем:

* Вызовы REST API своих сервисов.
* Чтение и запись в БД.
* Вызовы API других приложений.
* Чтение из очереди и запись туда.
* Запись в файловую систему.

Написание интеграционных тестов вокруг этих границ гарантирует, что запись данных и чтение данных от этих внешних участников работает нормально.  
  
При написании *узких интеграционных тестов* стремитесь локально запускать внешние зависимости: локальную базу данных MySQL, тест на локальной файловой системе ext4. Если интегрируетесь с отдельной службой, то или запустите экземпляр этой службы локально, или создайте и запустите поддельную версию, которая имитирует поведение реальной службы.  
  
Если нет возможности локально запустить стороннюю службу, то лучше запустить выделенный тестовый инстанс и указать на него в интеграционным тесте. В автоматизированных тестах избегайте интеграции с реальной системой продакшна. Запуск тысяч тестовых запросов на систему продакшна — верный способ разозлить людей, потому что вы забиваете их логи (в лучшем случае) или просто ддосите их сервис (в худшем случае). Интеграция с сервисом по сети — типичное свойство *широкого интеграционного теста*. Обычно из-за неё тесты труднее писать и они медленнее работают.  
  
Что касается пирамиды тестов, то интеграционные тесты находятся на более высоком уровне, чем модульные. Интеграция файловых систем и БД обычно гораздо медленнее, чем выполнение юнит-тестов с их имитациями. Их также труднее писать, чем маленькие изолированные модульные тесты. В конце концов, нужно думать о работе внешней части теста. Тем не менее, они имеют преимущество, потому что дают уверенность в правильной работе приложения со всеми внешними частями, с какими нужно. Юнит-тесты тут бесполезны.

### **Интеграция БД**

PersonRepository — единственный класс репозитория во всей кодовой базе. Он опирается на *Spring Data* и не имеет фактической реализации. Он просто расширяет интерфейс CrudRepository и предоставляет единственный заголовок метода. Остальное — магия Spring.

**public** **interface** **PersonRepository** **extends** **CrudRepository**<**Person**, **String**> {

Optional<Person> **findByLastName**(String lastName);

}

Через интерфейс CrudRepository Spring Boot предоставляет полностью функциональное хранилище CRUD с методами findOne, findAll, save, update и delete. Наше собственное определение метода findByLastName () расширяет эту базовую функциональность и даёт возможность получать людей, то есть объекты Person, по их фамилиям. Spring Data анализирует возвращаемый тип метода, имя метода и проверяет его на соответствие конвенциям именования для выяснения, что он должен делать.  
  
Хотя Spring Data выполняет большую работу по реализации репозиториев БД, я всё равно написал тест интеграции БД. Вы можете сказать, что это *тест фреймворка*, которого следует избегать, ведь мы тестируем чужой код. Тем не менее, я считаю, что здесь очень важно наличие хотя бы одного интеграционного теста. Во-первых, он проверяет нормальную работу нашего метода findByLastName. Во-вторых, это доказывает, что наш репозиторий правильно использует Spring и способен подключиться к БД.  
  
Чтобы облегчить выполнение тестов на вашем компьютере (без установки базы данных PostgreSQL), наш тест подключается к базе данных в памяти *H2*.  
  
Я определил H2 как тестовую зависимость в файле build.gradle. Файл application.properties в каталоге теста не определяет никаких свойств spring.datasource. Это указывает Spring Data использовать базу данных в памяти. Поскольку он находит H2 в пути к классу, то просто использует H2.  
  
Реальное приложение с профилем int (например, после установки в качестве переменной среды SPRING\_PROFILES\_ACTIVE=int) будет подключаться к базе данных PostgreSQL, как определено в application-int.properties.  
  
Понимаю, что здесь нужно знать и понимать кучу особенностей Spring. Придётся перелопатить [кучу документации](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-embedded-database-support). Финальный код простой с виду, но его трудно понять, если вы не знаете конкретных особенностей Spring.  
  
Кроме того, работа с базой данных в памяти — рискованное дело. В конце концов, наши интеграционные тесты работают с БД другого типа, чем в продакшне. Попробуйте и решите сами, предпочесть ли магию Spring и простой код — или явную, но более подробную реализацию.  
  
Ну, хватит объяснений. Вот простой интеграционный тест, который сохраняет объект Person в базу данных и находит его по фамилии.

@RunWith(SpringRunner.**class**)

@**DataJpaTest**

**public** **class** **PersonRepositoryIntegrationTest** {

@Autowired

**private** PersonRepository subject;

@After

**public** **void** **tearDown**() **throws** Exception {

subject.deleteAll();

}

@Test

**public** **void** **shouldSaveAndFetchPerson**() **throws** Exception {

Person peter = **new** Person("Peter", "Pan");

subject.save(peter);

Optional<Person> maybePeter = subject.findByLastName("Pan");

assertThat(maybePeter, is(Optional.of(peter)));

}

}

Как видите, наш интеграционный тест следует той же структуре «трёх А», что и юнит-тесты. Говорил же, что это универсальная концепция!

### **Интеграция с отдельными сервисами**

Наш микросервис получает погодные данные с darksky.net через REST API. Конечно, мы хотим убедиться, что сервис правильно отправляет запросы и разбирает ответы.  
  
При выполнении автоматических тестов желательно избежать взаимодействия с настоящими серверами *darksky*. Лимиты на нашем бесплатном тарифе — лишь одна из причин. Главное — это *отвязка*. Наши тесты должны запускаться независимо от того, какие справляются со своей работой милые люди в darksky.net. Даже если наша машина не может достучаться до серверов *darksky* или они закрылись на обслуживание.  
  
Чтобы избежать взаимодействия с реальными серверами *darksky*, мы для интеграционных тестов запускаем собственный, поддельный сервер *darksky*. Это может показаться очень трудной задачей. Но она упрощается благодаря таким инструментам, как [Wiremock](http://wiremock.org/). Смотрите сами:

@RunWith(SpringRunner.**class**)

@**SpringBootTest**

**public** **class** **WeatherClientIntegrationTest** {

@Autowired

**private** WeatherClient subject;

@Rule

**public** WireMockRule wireMockRule = **new** WireMockRule(8089);

@Test

**public** **void** **shouldCallWeatherService**() **throws** Exception {

wireMockRule.stubFor(get(urlPathEqualTo("/some-test-api-key/53.5511,9.9937"))

.willReturn(aResponse()

.withBody(FileLoader.read("classpath:weatherApiResponse.json"))

.withHeader(CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)

.withStatus(200)));

Optional<WeatherResponse> weatherResponse = subject.fetchWeather();

Optional<WeatherResponse> expectedResponse = Optional.of(**new** WeatherResponse("Rain"));

assertThat(weatherResponse, is(expectedResponse));

}

}

Для Wiremock создаем инстанс WireMockRule на фиксированном порту (8089). С помощью DSL можно настроить сервер Wiremock, определить конечные точки для прослушивания и предустановленные ответы.  
  
Далее вызываем тестируемый метод — тот, который обращается к сторонней службе — и проверяем, что результат правильно парсится.  
  
Важно понимать, как тест определяет, что должен обратиться к поддельному сервер Wiremock вместо реального API *darksky*. Секрет в файле application.properties, который располагается в src/test/resources. Его Spring загружает при выполнении тестов. В этом файле мы переопределяем конфигурацию вроде ключей API и URL-адресов со значениями, подходящими для тестов. В том числе назначаем вызов поддельного сервера Wiremock вместо реального:

weather.url = http://localhost:8089

Обратите внимание, что определённый здесь порт должен быть тем же, что мы указали при создании инстанса WireMockRule для теста. Замена URL-адреса реального API на поддельный стала возможной благодаря введению URL-адреса в конструктор класса WeatherClient:

@Autowired

**public** **WeatherClient**(**final** RestTemplate restTemplate,

@Value("${weather.url}") **final** String weatherServiceUrl,

@**Value**("${weather.api\_key}") **final** String weatherServiceApiKey) {

**this**.restTemplate = restTemplate;

**this**.weatherServiceUrl = weatherServiceUrl;

**this**.weatherServiceApiKey = weatherServiceApiKey;

}

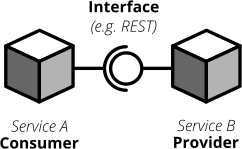
Так мы сообщаем нашему WeatherClient прочитать значение параметра weatherUrl свойства weather.url, которое мы определили в свойствах нашего приложения.  
  
С инструментами вроде Wiremock написание *узких интеграционных тестов* для отдельного сервиса становится достаточно простой задачей. К сожалению, у такого подхода есть недостаток: как гарантировать, что созданный нами поддельный сервер ведёт себя как настоящий? При текущей реализации отдельный сервис может изменить свой API, и наши тесты всё равно пройдут как ни в чём ни бывало. Сейчас мы просто тестируем, что WeatherClient способен воспринимать ответы от поддельного сервера. Это начало, но оно очень хрупкое. Проблему решают *сквозные тесты* и тестирование на реальном сервисе, но так мы становимся зависимы от его доступности. К счастью, есть лучшее решение этой дилеммы — контрактные тесты с участием и имитации, и реального сервера гарантируют, что имитация в наших интеграционных тестах точно соответствует оригиналу. Посмотрим, как это работает.

# **Контрактные тесты**

Более современные компании нашли способ масштабирования разработки путём распределения работ среди разных команд. Они создают отдельные, слабо связанные службы, не мешая друг на другу, и интегрируют их в большую, цельную систему. Именно с этим связана недавняя шумиха вокруг микросервисов.  
  
Разделение системы на множество небольших сервисов часто означает, что эти сервисы должны взаимодействовать друг с другом через определённые (желательно чётко определенные, но иногда случайно созданные) интерфейсы.  
  
Интерфейсы между разными приложения могут быть реализованы в разных форматах и технологиях. Самые распространённые:

* REST и JSON через HTTPS;
* RPC с использованием чего-то вроде [gRPC](https://grpc.io/);
* построение событийно-ориентированной архитектуры с использованием очередей.

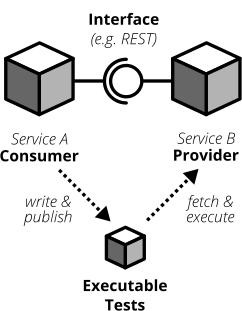
Каждый интерфейс задействует две стороны: поставщика и потребителя. **Поставщик** предоставляет данные потребителям. **Потребитель** обрабатывает данные, полученные от поставщика. В мире REST поставщик создаёт REST API со всеми необходимыми конечными точками, а потребитель обращается к этому REST API, чтобы получить данные или инициировать изменения в другой службе. В асинхронном мире событийно-ориентированной архитектуры поставщик (часто именуемый издателем) публикует данные в очередь; а потребитель (часто называемый подписчиком) подписывается на эти очереди, считывает и обрабатывает данные.



*Рис. 8. Каждый интерфейс задействует поставщика (или издателя) и потребителя (или подписчика). Спецификацией интерфейса можно считать контракт.*  
  
Поскольку сервисы поставщика и потребителя распределяются по разным командам, то вы оказываетесь в ситуации, когда нужно чётко указать интерфейс между ними (так называемый контракт). Традиционно компании подходят к этой проблеме следующим образом:

1. Написать длинную и подробную спецификацию интерфейса (*контракт*).
2. Реализовать сервис поставщика согласно определённому контракту.
3. Передать спецификации интерфейса стороне потребителя.
4. Подождать, пока они реализуют свою часть интерфейса.
5. Запустить крупномасштабный ручной системный тест, чтобы всё проверить.
6. Надеяться, что обе команды будут всегда соблюдать определения интерфейса и не облажаются.

Более современные компании заменили шаги 5 и 6 на автоматизированные [контрактные тесты](https://martinfowler.com/bliki/ContractTest.html), которые проверяют, что реализации на стороне потребителя и поставщика всё ещё придерживаются определённого контракта. Они выступают хорошим набором регрессионных тестов и гарантируют раннее обнаружение отклонения от контракта.  
  
В более гибкой организации следует выбрать более эффективный и менее расточительный маршрут. Приложение создаётся в рамках одной организации. Не должно быть проблемой переговорить с разработчиками других сервисов вместо того, чтобы забрасывать им чрезмерно подробную готовую документацию. В конце концов, это ваши сотрудники, а не сторонний вендор, с которым можно общаться только через службу поддержки клиентов или пуленепробиваемые юридические контракты.  
  
**Ориентированные на пользователя контрактные тесты** (CDC-тесты) позволяют [потребителям управлять реализацией контракта](https://martinfowler.com/articles/consumerDrivenContracts.html). С помощью CDC потребители пишут тесты, которые проверяют интерфейс для всех данных, которые им нужны. Затем команда публикует эти тесты, чтобы разработчики службы поставщика могли легко получить и запустить эти тесты. Теперь они могут разработать свой API, запустив тесты CDC. После прогона всех тестов они знают, что удовлетворили все потребности команды на стороне потребителя.



*Рис. 9. Контрактные тесты гарантируют, что поставщик и все потребители интерфейса придерживаются определённого контракта интерфейса. С помощью CDC-тестов потребители интерфейса публикуют свои требования в виде автоматизированных тестов; поставщики непрерывно получают и выполняют эти тесты*  
  
Такой подход позволяет команде поставщика интерфейса реализовать только то, что действительно необходимо (сохраняя простоту, YAGNI и всё такое). Команда поставщика должна непрерывно получать и выполнять эти CDC-тесты (в своём конвейере сборки), чтобы немедленно замечать любые критические изменения. Если они нарушают интерфейс, то их CDC-тесты не пройдут, предотвращая критические изменения. Пока тесты проходят, команда может вносить любые изменения, какие хочет, не беспокоясь о других командах. Ориентированный на потребителя контрактный подход сокращает процесс разработки до следующего:

1. Команда потребителя пишет автоматизированные тесты со всеми ожиданиями со стороны потребителей.
2. Они публикуют тесты для команды поставщика.
3. Команда поставщика непрерывно запускает CDC-тесты и следит за ними.
4. Команды немедленно вступают в переговоры, когда CDC-тесты ломаются.

Если ваша организация внедрит подход микросервисов, то проведение CDC-тестов — важный шаг к созданию автономных групп. CDC-тесты — это автоматизированный способ поощрить общение в коллективе. Они гарантируют, что интерфейсы между командами работают в любое время. Неудачный CDC-тест — хороший повод пойти к пострадавшей команде, поговорить о предстоящих изменениях API и выяснить, куда двигаться дальше.  
  
Наивная реализация CDC-тестов настолько же проста, как запросы к API и оценка ответов на предмет наличия всего необходимого. Затем эти тесты упаковываются в исполняемый файл (.gem, .jar, .sh) и загружаются куда-нибудь для другой команды (например, репозиторий вроде [Artifactory](https://www.jfrog.com/artifactory/)).  
  
В последние годы подход CDC становится более популярным и создано несколько инструментов для упрощения написания и обмена тестами.  
  
[Pact](https://github.com/realestate-com-au/pact), вероятно, самый известный среди них. Он предлагает утончённый подход к написанию тестов для потребителя и поставщика, предоставляет заглушки для отдельных служб и позволяет обмениваться CDC-тестами с другими командами. Pact портирован на множество платформ и может использоваться с языками JVM, Ruby, .NET, JavaScript и многими другими.  
  
Pact — разумный выбор, чтобы начать работу с CDC. [Документация](https://docs.pact.io/) сначала ошеломляет, но если набраться терпения, то её можно одолеть. Она помогает получить твёрдое понимание CDC, что в свою очередь облегчает вам задачу пропагандировать CDC для работы с другими командами.  
  
Ориентированные на пользователя контрактные тесты могут кардинально упростить работу автономных команд, которые начнут действовать быстро и уверенно. Сделайте одолжение, ознакомьтесь и попробуйте эту концепцию. Качественный набор CDC-тестов неоценим, чтобы быстро продолжать разработку, не ломая прочие сервисы и не огорчая другие команды.

### **Тест потребителя (наша команда)**

Наш микросервис использует погодный API. Так что наша обязанность — написать **тест потребителя**, который определяет наши ожидания по контракту (API) между нашим микросервисом и погодной службой.  
  
Сначала включаем библиотеку для написания тестов потребителя в нашу build.gradle:

testCompile('au.com.dius:pact-jvm-consumer-junit\_2.11:3.5.5')

Благодаря этой библиотеке мы можем реализовать тест потребителя и использовать сервисы имитации Pact:

@RunWith(SpringRunner.**class**)

@**SpringBootTest**

**public** **class** **WeatherClientConsumerTest** {

@Autowired

**private** WeatherClient weatherClient;

@Rule

**public** PactProviderRuleMk2 weatherProvider =

**new** PactProviderRuleMk2("weather\_provider", "localhost", 8089, **this**);

@Pact(consumer="test\_consumer")

**public** RequestResponsePact **createPact**(PactDslWithProvider builder) **throws** IOException {

**return** builder

.given("weather forecast data")

.uponReceiving("a request for a weather request for Hamburg")

.path("/some-test-api-key/53.5511,9.9937")

.method("GET")

.willRespondWith()

.status(200)

.body(FileLoader.read("classpath:weatherApiResponse.json"),

ContentType.APPLICATION\_JSON)

.toPact();

}

@Test

@PactVerification("weather\_provider")

**public** **void** **shouldFetchWeatherInformation**() **throws** Exception {

Optional<WeatherResponse> weatherResponse = weatherClient.fetchWeather();

assertThat(weatherResponse.isPresent(), is(**true**));

assertThat(weatherResponse.get().getSummary(), is("Rain"));

}

}

Если внимательно посмотреть, то WeatherClientConsumerTest очень похож на WeatherClientIntegrationTest. Только для серверной заглушки вместо Wiremock на этот раз мы используем Pact. На самом тест потребителя работает точно так же, как интеграционный: мы заменяем реальный сторонний сервер заглушкой, определяем ожидаемый ответ и проверяем, что клиент может правильно его разобрать. В этом смысле WeatherClientConsumerTest представляет собой узкий интеграционный тест. Преимущество по сравнению с тестом на основе Wiremock в том, что он при каждом запуске генерирует *файл Pact* (находится в target/pacts/&pact-name>.json). Он описывает наши ожидания по контракту в специальном формате JSON. Затем этот файл можно использовать для проверки, что сервер-заглушка ведёт себя как настоящий. Мы можем взять pact-файл и передать его команде, предоставляющей интерфейс. Они берут pact-файл и пишут тест провайдера, используя указанные там ожидания. Так они проверяют, соответствует ли их API всем нашим ожиданиям.  
  
Как можно понять, именно отсюда взялась часть *«ориентирование на потребителя»* в определении CDC. Потребитель управляет реализацией интерфейса, описывая свои ожидания. Поставщик должен убедиться, что он выполняет все ожидания. Никаких лишних спецификаций, YAGNI и все дела.  
  
Передача pact-файла в команду поставщика может пройти несколькими путями. Простой — зарегистрировать его в системе управления версиями и сказать команде поставщика всегда брать последнюю версию файла. Более продвинутый способ — использовать репозиторий артефактов вроде Amazon S3 или Pact Broker. Начинайте с простого и растите по мере необходимости.  
  
В реальном приложении вам не нужны одновременно и *интеграционный тест*, и *тест потребителя* для клиентского класса. Наш пример кода содержит оба только для демонстрации, как использовать каждый из них. Если вы хотите написать CDC-тесты на Pact, я рекомендую оставаться с ним. В этом случае преимущество в том, что вы автоматически получаете pact-файл с ожиданиями от контракта, которые другие команды могут использовать, чтобы легко сделать свои тесты поставщика. Конечно, это имеет смысл только если вы сможете убедить другую команду использовать Pact. Если нет, то используйте интеграционный тест интеграции в сочетании с Wiremock как достойную альтернативу.

### **Тест поставщика (другая команда)**

Тесты поставщика должны реализовать те, кто предоставляет погодный API. Мы используем публичный API от darksky.net. Теоретически, команда darksky со своей стороны должна выполнить тест поставщика и убедиться, что не нарушает контракт между своим приложением и нашим сервисом.  
  
Очевидно, что их не волнует наше скромное тестовое приложение — и они не будут делать для нас CDC-тест. Есть большая разница между публичным API и организацией, использующей микросервисы. Общедоступный API не может учитывать нужды каждого отдельного потребителя, иначе не сможет нормально работать. Внутри своей организации вы можете и должны их учитывать. Скорее всего, ваше приложение будет обслуживать несколько, ну может пару десятков потребителей. Ничего не мешает написать тесты поставщика для этих интерфейсов, чтобы сохранить стабильную систему.  
  
Команда поставщика получает pact-файл и запускает его на своём сервисе. Для этого она реализует тест, который считывает pact-файл, ставит несколько заглушек и проверяет на своём сервисе ожидания, определённые в pact-файле.  
  
Сообщество проекта Pact написало несколько библиотек для реализации тестов поставщика. В их [основном репозитории GitHub](https://github.com/DiUS/pact-jvm) неплохой выбор библиотек для потребителей и провайдеров. Выберите ту, которая лучше всего соответствует вашему стеку технологий.  
  
Для простоты предположим, что API darksky тоже реализован в Spring Boot. В этом случае они могут использовать [библиотеку Pact Spring](https://github.com/DiUS/pact-jvm/tree/master/pact-jvm-provider-spring), которая хорошо подключается к механизмам MockMVC Spring. Гипотетический тест поставщика, который могла бы реализовать команда darksky.net, выглядит так:

@RunWith(RestPactRunner.**class**)

@**Provider**("**weather\_provider**") // **same** **as** **the** "**provider\_name**" **in** **our** **clientConsumerTest**

@**PactFolder**("**target**/**pacts**") // **tells** **pact** **where** **to** **load** **the** **pact** **files** **from**

**public** **class** **WeatherProviderTest** {

@InjectMocks

**private** ForecastController forecastController = **new** ForecastController();

@Mock

**private** ForecastService forecastService;

@TestTarget

**public** **final** MockMvcTarget target = **new** MockMvcTarget();

@Before

**public** **void** **before**() {

initMocks(**this**);

target.setControllers(forecastController);

}

@State("weather forecast data") // same as the "given()" in our clientConsumerTest

**public** **void** **weatherForecastData**() {

when(forecastService.fetchForecastFor(any(String.**class**), **any**(**String**.**class**)))

.**thenReturn**(**weatherForecast**("**Rain**"));

}

}

Как видите, от поставщика требуется лишь загрузить pact-файл (например, @PactFolder определяет, откуда загружать полученные pact-файлы), а затем определить, как обеспечить тестовые данные для предопределённых состояний (например, с помощью имитаций Mockito). Не нужно писать какой-то специальный тест. Всё берётся из pact-файла. Важно, чтобы тест поставщика соответствовал *имени поставщика* и *состоянию*, объявленным в тесте потребителя.

### **Тест поставщика (наша команда)**

Мы посмотрели, как тестировать контракт между нашим сервисом и поставщиком погодной информации. В этом интерфейсе наш сервис выступает в качестве потребителя, а метеорологическая служба — в качестве поставщика. Подумав ещё, мы увидим, что наш сервис тоже выступает в роли поставщика для других: мы предоставляем REST API с несколькими конечными точками для других потребителей.  
  
Поскольку мы знаем важность контрактных тестов, то конечно напишем тест и для этого контракта. К счастью, наши контракты ориентированы на потребителя, так что все команды-потребители присылают нам свои pact-файлы, которые мы можем использовать для реализации тестов поставщика для нашего REST API.  
  
Сначала добавим в наш проект библиотеку поставщика Pact для Spring:  
  
testCompile('au.com.dius:pact-jvm-provider-spring\_2.12:3.5.5')  
  
Реализация теста поставщика следует той же описанной схеме. Для простоты я зарегистрирую pact-файл от нашего [простого потребителя](https://github.com/hamvocke/spring-testing-consumer) в репозитории нашего сервиса. Для нашего случая так проще, а в реальной жизни, вероятно, придётся использовать более сложный механизм для распространения pact-файлов.

@RunWith(RestPactRunner.**class**)

@**Provider**("**person\_provider**")// **same** **as** **in** **the** "**provider\_name**" **part** **in** **our** **pact** **file**

@**PactFolder**("**target**/**pacts**") // **tells** **pact** **where** **to** **load** **the** **pact** **files** **from**

**public** **class** **ExampleProviderTest** {

@Mock

**private** PersonRepository personRepository;

@Mock

**private** WeatherClient weatherClient;

**private** ExampleController exampleController;

@TestTarget

**public** **final** MockMvcTarget target = **new** MockMvcTarget();

@Before

**public** **void** **before**() {

initMocks(**this**);

exampleController = **new** ExampleController(personRepository, weatherClient);

target.setControllers(exampleController);

}

@State("person data") // same as the "given()" part in our consumer test

**public** **void** **personData**() {

Person peterPan = **new** Person("Peter", "Pan");

when(personRepository.findByLastName("Pan")).thenReturn(Optional.of

(peterPan));

}

}

Показанный ExampleProviderTest должен предоставить состояние в соответствии с полученным pact-файлом, вот и всё. Когда мы запустим тест, Pact подберёт pact-файл и отправит HTTP-запрос к нашему сервису, который ответит в соответствии с заданным состоянием.

# **Тесты UI**

У большинства приложений есть какой-то пользовательский интерфейс. Обычно мы говорим о веб-интерфейсе в контексте веб-приложений. Люди часто забывают, что REST API или интерфейс командной строки — это такой же UI, как и причудливый веб-интерфейс.  
  
*Тесты UI* проверяют правильность работы пользовательского интерфейса приложения. Действия пользователя должны инициировать правильные события, данные должны представляться пользователю, состояние UI должно изменяться ожидаемым образом.



Иногда говорят, что тесты UI и сквозные тесты — это одно и то же (как говорит Майк Кон). Для меня это отождествление двух вещей с весьма ортогональными концепциями.  
  
Да, тестирование приложения от начала до конца часто означает прохождение через пользовательский интерфейс. Но обратное неверно.  
  
Тестирование пользовательского интерфейса необязательно должно проводиться в сквозном режиме. В зависимости от используемой технологии, тестирование UI может оказаться таким же простым, как написание некоторых модульных тестов для фронтенда JavaScript с заглушенным бэкендом.  
  
Для тестирования UI традиционных веб-приложений предназначены специальные инструменты вроде [Selenium](http://docs.seleniumhq.org/). Если вы считаете пользовательским интерфейсом REST API, то достаточно правильных интеграционных тестов вокруг API.  
  
В веб-интерфейсах желательно проверить несколько аспектов UI, в том числе поведение, вёрстка, юзабилити, соблюдение фирменного стиля и др.  
  
К счастью, тестирование поведения UI довольно простое. Щёлкаете здесь, вводите данные там — и проверяете, что состояние UI меняется соответствующим образом. Современные фреймворки для одностраничных приложений ([react](https://facebook.github.io/react/), [vue.js](https://vuejs.org/), [Angular](https://angular.io/) и прочие) часто поставляются с инструментами и хелперами для тщательного тестирования этих взаимодействий на довольно низком уровне (в юнит-тесте). Даже если выкатить собственную реализацию фронтенда на ванильном JavaScript, всё равно можно использовать обычные инструменты тестирования, такие как [Jasmine](https://jasmine.github.io/) и [Mocha](http://mochajs.org/). Для более традиционного приложения с рендерингом на стороне сервера наилучшим выбором станут тесты на основе Selenium.  
  
Цельность *вёрстки* веб-приложения проверить немного сложнее. В зависимости от приложения и потребностей пользователей может возникнуть необходимость убедиться, что изменения кода случайно не нарушают вёрстку сайта.  
  
Но компьютеры плохо справляются с проверкой, что всё «нормально выглядит» (возможно, в будущем какой-то умный алгоритм машинного обучения изменит это).  
  
Есть некоторые инструменты, чтобы попробовать автоматическую проверку дизайна веб-приложения в конвейере сборки. Большинство из них используют Selenium для открытия веб-приложения в разных браузерах и форматах, произведения скриншотов и сравнения с ранее сделанными скриншотами. Если старый и новый скриншоты отличаются неожиданным образом, то инструмент подаст сигнал.  
  
Один из таких инструментов — [Galen](http://galenframework.com/). Некоторые команды используют [lineup](https://github.com/otto-de/lineup) и его брата jlineup на основе Java для достижения аналогичного результата. Оба инструмента применяют тот же подход на основе Selenium.  
  
Как только вы хотите проверить *удобство использования* и *приятный дизайн* — вы покидаете пространство автоматизированного тестирования. Здесь придётся полагаться на [исследовательские тесты](https://en.wikipedia.org/wiki/Exploratory_testing), тесты юзабилити (вплоть до простейших [холл-тестов](https://en.wikipedia.org/wiki/Usability_testing#Hallway_testing) на случайных людях). Придётся проводить демонстрации пользователям и проверять, нравится ли им продукт и могут ли они использовать все функции без разочарования или раздражения.

# **Сквозные тесты**

Тестирование развёрнутого приложения через UI — это самый полный тест, какой только можно провести. Описанные выше тесты UI через WebDriver — хорошие примеры сквозных тестов.



*Рис. 11. Сквозные тесты проверяют полностью интегрированную систему целиком*  
  
Сквозные тесты (также называемые [тестами широкого стека](https://martinfowler.com/bliki/BroadStackTest.html)) дают максимальную уверенность, работает программное обеспечение или нет. [Selenium](http://docs.seleniumhq.org/) и [протокол WebDriver](https://www.w3.org/TR/webdriver/) позволяют автоматизировать тесты, автоматически отправляя headless-браузер на развёрнутые сервисы для выполнения кликов, ввода данных и проверки состояния UI. Можно использовать Selenium напрямую или применить инструменты на его основе, такие как [Nightwatch](http://nightwatchjs.org/).  
  
У сквозных тесты другие проблемы. Они известны своей ненадёжностью, сбоями по неожиданным и непредвиденным причинам. Довольно часто это ложноположительные сбои. Чем более сложный UI, тем более хрупкими становятся тесты. Причуды браузера, проблемы с синхронизацией, анимация и неожиданные всплывающие диалоги — лишь некоторые из причин, из-за которых я потратил больше времени на отладку, чем хотелось бы.  
  
В мире микросервисов также непонятно, кто отвечает за написание этих тестов. Поскольку они охватывают несколько сервисов (всю систему), то нет одной конкретной команды, ответственной за написание сквозных тестов.  
  
Если есть централизованная команда *обеспечения качества*, они выглядят хорошим кандидатом. Опять же, заводить централизованную команду QA строго не рекомендуется, такого не должно быть в мире DevOps, где все команды по-настоящему универсальны. Нет простого ответа, кто должен владеть сквозными тестами. Может, в вашей организации есть какая-то инициативная группа или *гильдия качества*, чтобы позаботиться о них. Здесь многое зависит от конкретной организации.  
  
Кроме того, сквозные тесты требуют серьёзной поддержки и выполняются довольно медленно. Если у вас много микросервисов, то вы даже не сможете запускать сквозные тесты локально, потому что тогда понадобится и все микросервисы запускать локально. Попробуйте запустить сотни приложений на своём компьютере, тут никакой оперативки не хватит.  
  
Из-за высоких расходов на обслуживание следует свести число сквозных тестов к абсолютному минимуму.  
  
Подумайте о самых главных взаимодействиях пользователей с приложением. Придумайте главные «маршруты» пользователей от экрана к экрану, чтобы автоматизировать самые важные из этих шагов в сквозных тестах.  
  
Если вы делаете интернет-магазин, то самым ценным «маршрутом» будет поиск продукта — помещение его в корзину — оформление заказа. Вот и всё. Пока этот маршрут работает, нет особых проблем. Возможно, вы найдёте еще пару важных маршрутов для сквозных тестов. Всё остальное, вероятно, принесёт больше проблем, чем пользы.  
  
Помните: в вашей пирамиде тестов много низкоуровневых тестов, где мы уже протестировали все варианты пограничных ситуаций и интеграции с другими частями системы. Нет необходимости повторять эти тесты на более высоком уровне. Большие усилия по техническому обслуживанию и много ложных срабатываний слишком замедлит вашу работу, а рано или поздно лишит вас доверия к тестам вообще.

### **Сквозные тесты UI**

Для сквозных тестов многие разработчики выбирают [Selenium](http://docs.seleniumhq.org/) и протокол [WebDriver](https://www.w3.org/TR/webdriver/). С Selenium можете выбрать любой браузер и натравить его на сайт. Пусть нажимает повсюду кнопки и ссылки, вводит данные и проверяет изменения в UI.  
  
К Selenium нужен браузер, который можно запустить и использовать для тестов. Есть несколько так называемых *«драйверов»* к разным браузерам. Выберите один (или несколько) и добавьте его в свой build.gradle. Какой бы браузер вы ни выбрали, следует убедиться, что у всех разработчиков и на сервере CI установлена правильная версия браузера. Может оказаться трудно обспечить такую синхронизацию. Для Java есть небольшая библиотека [webdrivermanager](https://github.com/bonigarcia/webdrivermanager), которая автоматизирует загрузку и настройку правильной версии браузера. Добавьте две такие зависимости в build.gradle:

testCompile('org.seleniumhq.selenium:selenium-chrome-driver:2.53.1')

testCompile('io.github.bonigarcia:webdrivermanager:1.7.2')

Запуск полноценного браузера в тестовом наборе может стать проблемой. Особенно если сервер непрерывной доставки, где работает наш конвейер, не способен развернуть браузер с UI (например, потому что X-Server недоступен). В этом случае можно запустить виртуальный X-Server вроде [xvfb](https://en.wikipedia.org/wiki/Xvfb).  
  
Более новый подход заключается в использовании *headless*-браузера (т.е. браузера без пользовательского интерфейса) для тестов WebDriver. До недавнего времени чаще всего для автоматизации браузерных задач использовался PhantomJS. Но когда [Chromium](https://developers.google.com/web/updates/2017/04/headless-chrome) и [Firefox](https://developer.mozilla.org/en-US/Firefox/Headless_mode) внедрили headless-режим из коробки, PhantomJS внезапно устарел. В конце концов, лучше протестировать сайт с помощью реального браузера, который действительно есть у пользователей (например, Firefox и Chrome), а не с помощью искусственного браузера только потому что это удобно вам как разработчику.  
  
Оба headless-браузера Firefox и Chrome совершенно новые и ещё не получили широкого распространения для тестов WebDriver. Мы не хотим ничего усложнять. Вместо возни со свеженькими headless-режимами давайте придерживаться классического способа, то есть Selenium в связке с обычным браузером. Вот как выглядит простой сквозной тест, который запускает Chrome, переходит к нашему сервису и проверяет содержимое сайта:

@RunWith(SpringRunner.**class**)

@**SpringBootTest**(**webEnvironment** = SpringBootTest.WebEnvironment.RANDOM\_PORT)

**public** **class** **HelloE2ESeleniumTest** {

**private** WebDriver driver;

@LocalServerPort

**private** **int** port;

@BeforeClass

**public** **static** **void** **setUpClass**() **throws** Exception {

ChromeDriverManager.getInstance().setup();

}

@Before

**public** **void** **setUp**() **throws** Exception {

driver = **new** ChromeDriver();

}

@After

**public** **void** **tearDown**() {

driver.close();

}

@Test

**public** **void** **helloPageHasTextHelloWorld**() {

driver.get(String.format("http://127.0.0.1:%s/hello", port));

assertThat(driver.findElement(By.tagName("body")).getText(), containsString("Hello World!"));

}

}

Обратите внимание, что этот тест будет работать, только если Chrome установлен на машине, где запускается тест (ваш локальный компьютер, сервер CI).  
  
Тест простой. Он запускает приложение Spring на случайном порту с помощью @SpringBootTest. Затем создаётся новый «веб-драйвер» Chrome, он получает указание перейти к конечной точке /hello нашего микросервиса и проверить, что в окне браузера печатается «Hello World!». Классно!

### **Сквозной тест REST API**

Для повышения надёжности тестов хорошая идея — избегать GUI. Такие тесты более стабильны, чем полноценные сквозные тесты, и в то же время покрывают значительную часть стека приложения. Это может пригодиться, если тестировать приложение через веб-интерфейс особенно сложно. Может, у вас даже нет веб-интерфейса, а только REST API (потому что одностраничное приложение где-то общается с этим API или просто потому что вы презираете всё красивое и блестящее). В любом случае ситуация подходит для [подкожного теста](https://martinfowler.com/bliki/SubcutaneousTest.html) (subcutaneous test), который тестирует всё, что находится под GUI. Если вы обслуживаете REST API, то такой тест будет правильным, как в нашем примере:

@RestController

**public** **class** **ExampleController** {

**private** **final** PersonRepository personRepository;

// shortened for clarity

@GetMapping("/hello/{lastName}")

**public** String **hello**(@PathVariable **final** String lastName) {

Optional<Person> foundPerson = personRepository.findByLastName(lastName);

**return** foundPerson

.map(person -> String.format("Hello %s %s!",

person.getFirstName(),

person.getLastName()))

.orElse(String.format("Who is this '%s' you're talking about?",

lastName));

}

}

Позвольте показать ещё одну библиотеку, которая пригодится при тестировании сервиса, предоставляющего REST API. Библиотека [REST-assured](https://github.com/rest-assured/rest-assured) даёт хороший DSL для запуска реальных HTTP-запросов к API и оценки полученных ответов.  
  
Прежде всего, добавьте зависимость в свой build.gradle.

testCompile('io.rest-assured:rest-assured:3.0.3')

С помощью этой библиотеки можно реализовать сквозной тест для нашего REST API:

@RunWith(SpringRunner.**class**)

@**SpringBootTest**(**webEnvironment** = SpringBootTest.WebEnvironment.RANDOM\_PORT)

**public** **class** **HelloE2ERestTest** {

@Autowired

**private** PersonRepository personRepository;

@LocalServerPort

**private** **int** port;

@After

**public** **void** **tearDown**() **throws** Exception {

personRepository.deleteAll();

}

@Test

**public** **void** **shouldReturnGreeting**() **throws** Exception {

Person peter = **new** Person("Peter", "Pan");

personRepository.save(peter);

when()

.get(String.format("http://localhost:%s/hello/Pan", port))

.then()

.statusCode(is(200))

.body(containsString("Hello Peter Pan!"));

}

}

Мы опять запускаем всё приложение Spring с помощью @SpringBootTest. В этом случае мы делаем @Autowire для PersonRepository, чтобы легко записывать тестовые данные в БД. Теперь когда просим REST API сказать «привет» нашему другу “Mr Pan”, то видим приятное приветствие. Потрясающе! И более чем достаточно для сквозного теста, если веб-интерфейс вообще отсутствует.

# **Приёмочные тесты — ваши фичи правильно работают?**

Чем выше вы поднимаетесь в пирамиде тестов, тем больше вероятность возникновения вопросов: правильно ли работают фичи с точки зрения пользователя. Вы можете рассматривать приложение как чёрный ящик и изменить направленность тестов от прошлого:

когда я ввожу x и y, возвращаемое значение должно быть z

к следующему:

*учитывая*, что есть авторизованный пользователь  
  
*и* есть изделие «велосипед»  
  
*когда* пользователь переходит на страницу описания изделия «велосипед»  
  
*и* нажимает кнопку «Добавить в корзину»  
  
*тогда* изделие «велосипед» должно быть в его корзине

Бывает, что такие тесты называют [функциональными](https://en.wikipedia.org/wiki/Functional_testing) или [приёмочными](https://en.wikipedia.org/wiki/Acceptance_testing#Acceptance_testing_in_extreme_programming). Некоторые говорят, что функциональные и приёмочные тесты — это разные вещи. Иногда термины объединяют. Иногда люди бесконечно спорят о формулировках и определениях. Часто такие обсуждения вводят ещё бóльшую путаницу.  
  
Вот что: в какой-то момент следует убедиться, что ваша программа правильно работает с точки зрения *пользователя*, а не только с технической точки зрения. Как вы называете эти тесты — на самом деле не так важно. А вот наличие этих тестов важно. Выберите любой термин, придерживайтесь его и напишите эти тесты.  
  
Можно ещё упомянуть BDD (разработка, основанная на описании поведения) и инструменты для нее. BDD и соответствующий стиль написания тестов — хороший трюк, чтобы изменить своё мышление от деталей реализации к потребностям пользователей. Не бойтесь и попробуйте.  
  
Вам даже не обязательно внедрять полномасштабные инструменты BDD, такие как [Cucumber](https://cucumber.io/) (хотя можно и внедрить). Некоторые assertion-библиотеки вроде [chai.js](http://chaijs.com/guide/styles/#should) позволяют писать утверждения (assertions) с ключевыми словами в стиле should, которые приближают тесты к BDD. И даже если вы не используете библиотеку, которая предоставляет такую нотацию, умный и хорошо продуманный код позволит писать тесты, ориентированные на поведение пользователей. Некоторые методы/функции хелперов могут оказаться очень успешными:

# a sample acceptance test in Python

**def** **test\_add\_to\_basket**():

# given

user = a\_user\_with\_empty\_basket()

user.login()

bicycle = article(name="bicycle", price=100)

# when

article\_page.add\_to\_.basket(bicycle)

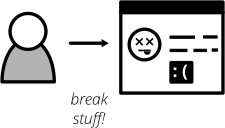
# then

**assert** user.basket.contains(bicycle)

Приёмочные тесты могут проводиться на разных уровнях детализации. В основном они будут достаточно высокого уровня и тестировать сервис через UI. Но важно понимать, что технически нет обязательного требования писать приёмочные тесты именно на самом высоком уровне пирамиды тестов. Если структура приложения и имеющийся сценарий позволяют написать приёмочный тест на более низком уровне, сделайте это. Тест низкого уровня лучше, чем высокого. Концепция приёмочных тестов — доказать, что фичи приложения правильно работают для пользователя — полностью ортогональна вашей пирамиде тестов.

# **Исследовательское тестирование**

Даже самые прилежные усилия по автоматизации тестов не идеальны. Иногда в автоматических тестах вы пропускаете определённые пограничные случаи. Иногда просто невозможно обнаружить определённую ошибку, написав модульный тест. Некоторые проблемы качества вообще не проявятся в автоматизированных тестах (подумайте о дизайне или юзабилити). Несмотря на самые лучшие намерения в отношении автоматизации тестов, ручные тесты в некоторых отношениях по-прежнему незаменимы.



*Рис. 12. Исследовательское тестирование выявит проблемы качества, незамеченные в процессе сборки*  
  
Включите [исследовательские тесты](https://en.wikipedia.org/wiki/Exploratory_testing) в свой набор тестов. Эта процедура тестирования вручную подчёркивает свободу и творческие способности тестировщика, который способен найти проблемы качества в работающей системе. Просто выделите немного времени в расписании, засучите рукава и постарайтесь вызвать сбой приложения каким-нибудь способом. Включите деструктивное мышление и придумывайте способы, как спровоцировать проблемы и ошибки в программе. Документируйте всё, что найдёте. Ищите баги, проблемы дизайна, медленное время отклика, отсутствующие или вводящие в заблуждение сообщения об ошибках и всё остальное, что вас раздражает как пользователя.  
  
Хорошая новость в том, что вы легко можете автоматизировать тесты для большинства найденных ошибок. Написание автоматизированных тестов для найденных ошибок гарантирует, что в будущем не будет регрессий этой ошибки. Кроме того, это помогает выяснить корневую причину проблемы при исправлении бага.  
  
Во время исследовательского тестирования вы обнаружите проблемы, которые незаметно проскользнули через конвейер сборки. Не расстраивайтесь. Это хорошая обратная связь для улучшения конвейера сборки. Как и с любой обратной связью, обязательно отреагируйте со своей стороны: подумайте, какие действия предпринять, чтобы избежать такого рода проблем в будущем. Может, вы пропустили определённый набор автоматических тестов. Возможно, были небрежны с автоматизированными тестами на этом этапе и следует более тщательно проводить тесты в будущем. Возможно, есть какой-то блестящий новый инструмент или подход, который можно использовать в своем конвейере, чтобы избежать таких проблем. Обязательно отреагируйте так, чтобы ваш конвейер и вся система поставки программного обеспечения стали лучше и совершенствовались с каждым шагом.

# **Путаница с терминологией в тестировании**

Всегда сложно говорить о разных классификациях тестов. Моё понимание *юнит-тестов* (модульных тестов) может слегка отличаться от вашего. С интеграционными тестами ещё хуже. Для некоторых людей интеграционное тестирование — это очень широкая деятельность, которая тестирует множество различных частей всей системы. Для меня это довольно узкая вещь: тестирование только интеграции с одной внешней частью за раз. Некоторые называют это *интеграционными* тестами, некоторые — *компонентными*, другие предпочитают термин *сервисный тест*. Кто-то заявит, что это вообще три совершенно разные вещи. Нет правильного или неправильного определения. Сообщество разработчиков ПО просто не установило чётко определённых терминов в тестировании.  
  
Не зацикливайтесь на двусмысленных терминах. Не имеет значения, называете вы это сквозным тестом, тестом широкого стека или функциональным тестом. Неважно, если ваши интеграционные тесты означают для вас не то, что для людей в другой компании. Да, было бы очень хорошо, если бы наша отрасль могла чётко определить термины и все бы их придерживались. К сожалению, этого ещё не произошло. И поскольку в тестировании много нюансов, то всё равно мы имеем дело скорее с широким спектром тестов, чем с кучей дискретных множеств, что ещё больше усложняет чёткую терминологию.  
  
Важно воспринимать это так: вы просто находите термины, которые работают для вас и вашей команды. Ясно определите для себя различные типы тестов, которые хотите написать. Согласуйте термины в своей команде и найдите консенсус относительно охвата каждого типа теста. Если в своей команде (или даже во всей организации) вы будете последовательны с этими терминами, то это всё, о чем нужно позаботиться. [Саймон Стюарт](https://testing.googleblog.com/2010/12/test-sizes.html) хорошо подытожил это в подходе, который используется в Google. Думаю, это прекрасная демонстрация, что не стоит слишком зацикливаться на названиях и конвенциях по терминам.

# **Внедрение тестов в конвейер развёртывания**

Если вы используете непрерывную интеграцию или непрерывную доставку, то ваш [конвейер развёртывания](https://martinfowler.com/bliki/DeploymentPipeline.html) запускает автоматические тесты каждый раз при внесении изменений в ПО. Обычно конвейер разделяется на несколько этапов, которые постепенно дают всё больше уверенности, что ваша программа готова к развёртыванию в рабочей среде. Услышав обо всех разновидностях тестов, возможно, вы задаётесь вопросом, как поместить их в конвейер развёртывания. Чтобы ответить на это, следует просто подумать об одной из самых основополагающих ценностей непрерывной доставки (это одна из ключевых ценностей [экстремального программирования](http://www.extremeprogramming.org/values.html) и гибкой разработки): о **быстрой обратной связи**.  
  
Хороший конвейер сборки максимально быстро сообщает об ошибках. Вы не хотите ждать целый час, чтобы узнать, что последнее изменение сломало некоторые простые модульные тесты. Если конвейер работает так медленно, то вы могли уже уйти домой, когда поступила обратная связь. Информация должна приходить в течение нескольких секунд или нескольких минут с быстрых тестов на ранних этапах конвейера. И наоборот, более длительные тесты — обычно с более широкой областью — размещаются на более поздних этапах, чтобы не тормозить фидбек от быстрых тестов. Как видите, этапы конвейера развёртывания определяются не типами тестов, а их скоростью и областью действия. Поэтому очень разумно может быть разместить некоторые из самых узких и быстрых интеграционных тестов на ту же стадию, что и юнит-тесты — просто потому что они дают более быструю обратную связь. И необязательно проводить строгую линию по формальному типу тестов.

# **Избегайте дублирования тестов**

Есть ещё одна ловушка, которую следует избегать: дублирование тестов на разных уровнях пирамиды. Чутьё говорит, что тестов много не бывает, но позвольте вас заверить: бывает. Каждый тест в тестовом наборе — дополнительный багаж, который не обходится бесплатно. Написание и ведение тестов требует времени. Чтение и понимание чужого теста требует времени. И конечно, выполнение тестов тоже требует времени.  
  
Как и с производственным кодом, следует стремиться к простоте и избегать дублирования. В контексте реализации пирамиды тестов есть два эмпирических правила:

1. Если в тесте более высокого уровня обнаружена ошибка, а в тестах более низкого уровня нет, то необходимо писать тест более низкого уровня.
2. Сдвигайте тесты как можно ниже по уровням пирамиды.

Первое правило важно, потому что тесты более низкого уровня лучше позволяют сузить область и изолированно воспроизвести ошибку. Они работают быстрее и менее раздуты, что помогает при отладке вручную. И в будущем послужат хорошим регрессионным тестом. Второе правило важно для быстрого выполнения набора тестов. Если вы уверенно протестировали все условия на тестах более низкого уровня, то в тесте более высокого уровня нет необходимости. Он просто не добавляет уверенности, что всё работает. Избыточные тесты станут обузой и начнут раздражать в повседневной работе. Набор тестов будет работать медленнее, и при изменении кода потребуется изменить больше тестов.  
  
Сформулируем иначе: если тест более высокого уровня даёт больше уверенности, что приложение работает правильно, то нужно иметь такой тест. Написание модульного теста для класса контроллера помогает проверить логику внутри самого контроллера. Тем не менее, это не скажет вам, действительно ли конечная точка REST, которую предоставляет этот контроллер, отвечает на запросы HTTP. Таким образом, вы продвигаетесь вверх по пирамиде тестов и добавляете тест, который проверяет именно это, но не более того. В тесте более высокого уровня вы не тестируете всю условную логику и пограничные случаи, которые уже покрыты юнит-тестами более низкого уровня. Убедитесь, что тест высокого уровня фокусируется только на том, что не покрыто тестами более низкого уровня.  
  
Я строго отношусь к исключению тестов, не имеющих ценности. Удаляю высокоуровневые тесты, которые уже покрыты на более низком уровне (учитывая, что они не дают дополнительной ценности). Заменяю тесты более высокого уровня тестами более низкого уровня, если это возможно. Иногда трудно удалить лишний тест, особенно если придумать его было непросто. Но вы рискуете создать [невозвратные затраты](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B2%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82%D0%BD%D1%8B%D0%B5_%D0%B7%D0%B0%D1%82%D1%80%D0%B0%D1%82%D1%8B), так что смело жмите Delete. Нет причин тратить драгоценное время на тест, который перестал приносить пользу.

# **Пишите чистый код для тестов**

Как и в отношении обычного кода, следует позаботиться о хорошем и чистом коде тестов. Вот ещё несколько советов для создания поддерживаемого тестового кода, прежде чем начинать работу и создавать автоматизированный набор тестов:

1. Тестовый код так же важен, как и код в продакшне. Уделите ему столько же заботы и внимания. «Это всего лишь тест» — не оправдание для небрежного кода.
2. Проверяйте только одно условие в каждом тесте. Тогда тесты останутся лаконичными и понятными.
3. Правило трёх А (arrange, act, assert) или триада «дано, когда, тогда» — хорошая мнемоника, чтобы поддерживать хорошую структуру тестов.
4. Читаемость имеет значение. Не злоупотребляйте DRY (правило «не повторяйся»). Повторение хорошо, если улучшает читаемость. Попробуйте найти баланс между кодом [DRY и DAMP](https://stackoverflow.com/questions/6453235/what-does-damp-not-dry-mean-when-talking-about-unit-tests) (DAMP — Descriptive And Meaningful Phrases, содержательные и осмысленные фразы).
5. Если сомневаетесь насчёт рефакторинга или повторного использования кода, применяйте [Правило Трёх](https://blog.codinghorror.com/rule-of-three/). *Use before reuse*.

# **Заключение**

Вот и всё! Знаю, это было долгое и трудное объяснение, почему и как следует проводить тестирование. Отличная новость в том, что эта информация практически не имеет срока давности и не зависит от того, какую программу вы создаёте. Работаете вы над микросервисами, устройствами IoT, мобильными приложениями или веб-приложениями, уроки из этой статьи применимы ко всему.  
  
Надеюсь, что в статье есть что-то полезное. Теперь вперёд, изучите [пример кода](https://github.com/hamvocke/spring-testing) и примените усвоенные понятия в своём наборе тестов. Создание крепкого набора тестов требует определённых усилий. Это окупится в долгосрочной перспективе и сделает вашу жизнь разработчика более спокойной, поверьте мне.