GeekBrains

Зависимости в тестах

Введение в юнит-тестирование

Оглавление

Введение

На прошлом уроке мы обсудили критерии качества тестов, тестирование по принципу чёрного и белого ящика, метрики тестов и инструменты для их измерения, TDD и BDD.

На этой лекции вы узнаете:

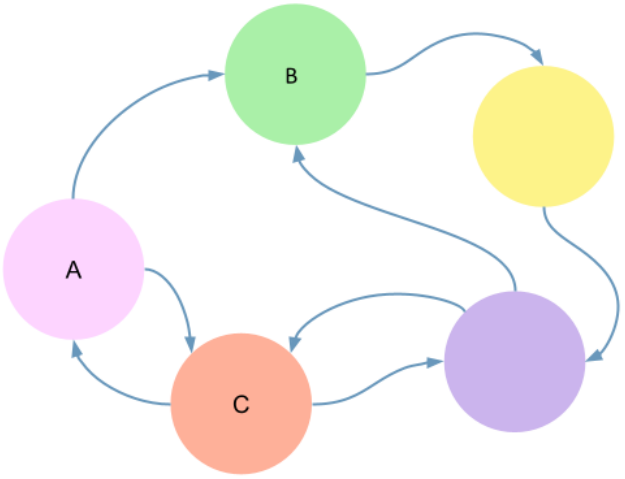
- Что такое зависимости.
- Виды зависимостей.
- Тестовые заглушки (Test Double).
- Подходы к изоляции кода в школах тестирования.
- Типы тестовых заглушек.
- Фреймворк Mockito.
- Создание и настройка моков с Mockito.
- Другие mocking-фреймворки.

Термины, используемые в лекции

Docker-контейнер — развёрнутое и запущенное в изолированном окружении приложение в виде специального образа.

Что такое зависимости

У большинства классов, с которыми мы сталкиваемся, когда пишем юнит-тесты, есть зависимости. Чаще всего классы не существуют изолированно в вакууме.



На рисунке условно показаны связи между классами абстрактной системы. Зависимости имеют направленность. То, что A зависит от B, не значит, что B зависит от A. Когда класс A использует класс или интерфейс B, тогда A зависит от B. Класс A не может выполнить свою работу без класса B, и так же один класс не может быть переиспользован без переиспользования другого. В таких случаях класс A называют «зависимым», а класс или интерфейс B — «зависимостью».

Ниже — утрированный код некоторого класса-сервиса, работающего с базой данных. Классы связываются в конструкторе сервиса таким образом, что состояние класса Service напрямую зависит от состояния класса Database. Это пример зависимости.

```
public class Service {

    private Database database;

    public Service(Database database) {
        this.database = database;
    }
}
```

💡 Для достижения целей модульного тестирования мы должны уметь контролировать поведение зависимостей внутри тестов.

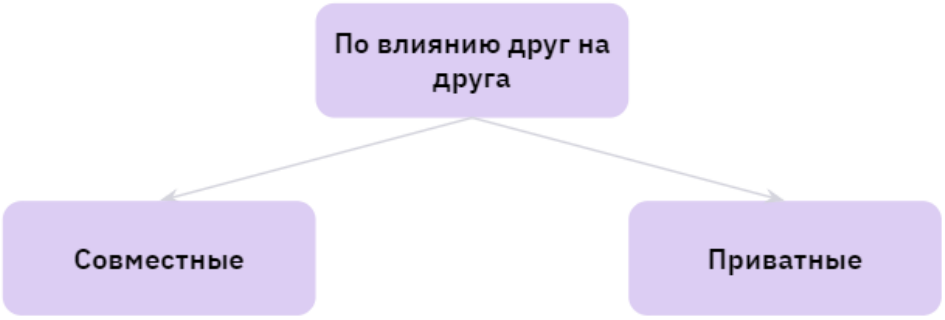
Типы зависимостей

Есть несколько классификаций зависимостей.

С точки зрения изменяемости зависимости делятся на изменяемые и неизменяемые.

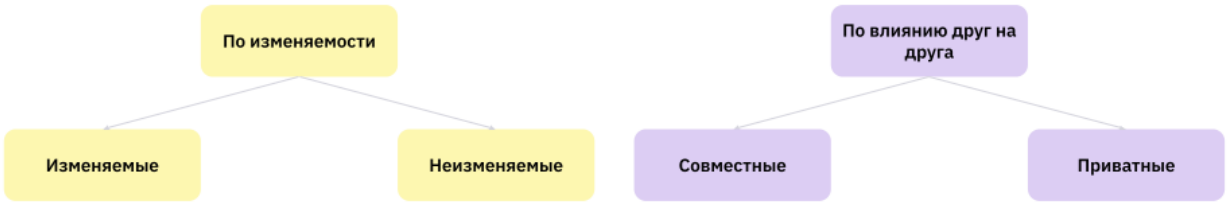
- **Изменяемые (muttable)** — зависимости, состояние которых может изменить тест: например, переменные или база данных.
- **Неизменяемые (immutable)** — зависимости, на которые не может повлиять тест: например, константа.

С точки зрения возможности тестов влиять друг на друга зависимости делятся на совместные и приватные.



- **Совместная (shared)** — зависимость, доступ к которой есть более чем у одного теста, и которая предоставляет им возможность влиять на результаты друг друга. Типичный пример — статическое изменяемое поле. Изменение в нём отражается на всех юнит-тестах, выполняемых в одном процессе.
- **Приватная (private)** — зависимость, которая не является совместной. Может быть как изменяемой, так и неизменяемой.

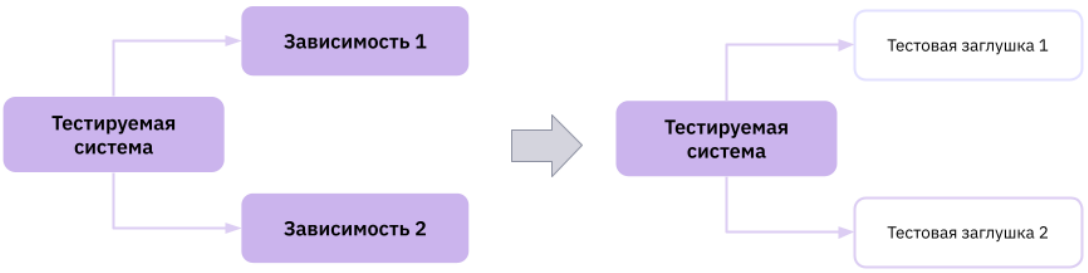
База данных может быть как совместной, так и приватной зависимостью. Совместной, когда все тесты работают с одной базой, приватной — когда для каждого теста поднимается отдельная база данных, например в docker-контейнере. В этом случае тесты не смогут повлиять друг на друга через базу.



Тестовые заглушки (test double)

Когда в киноиндустрии снимают что-то рискованное для актёра, нанимают дублёра-каскадёра — специалиста, который сумеет справиться с опасной задачей. Возможно, он не умеет играть на сцене, но знает, как упасть с большой высоты или разбить машину. **Насколько дублёр должен быть похож на актёра, зависит от сцены.** Часто достаточно кого-то отдалённо похожего по росту.

Так же поступают и в тестах с зависимостями — их заменяют на **тестовые заглушки (test double).** Это общий термин, который описывает все разновидности фиктивных зависимостей в тестах. Английский вариант термина происходит от stunt double — дублёров на съёмках.



Тестовые заглушки используют для упрощения тестирования: они передаются тестируемой системе вместо реальных зависимостей, настройка и сопровождение которых могут быть сопряжены с определёнными сложностями.

Термин «тестовая заглушка» ввёл Джерард Месарош в книге «xUnit Test Patterns: Refactoring Test Code».

> 💡 Тестовые заглушки настраиваются таким образом, чтобы гарантировать правильность их работы. Если тест не проходит, мы можем быть уверены, что ошибка не связана с тестовым дубликатом, и мы можем искать проблему в другом месте.

Важно: используя заглушки, мы получаем тест, который проверяет тестируемую систему без зависимостей, мешающих его работе. Но полностью изолировать тест от всех зависимостей — плохая идея. Нужно осознанно подходить к выбору зависимостей, которые мы заменяем заглушками.

Что заменять тестовыми заглушками?

Юнит-тест — это автоматизированный тест, который проверяет правильность работы юнит-единицы кода, проводит тестирование быстро и **изолирован от другого кода.**

Существуют большие расхождения во мнениях по поводу третьего атрибута. Из-за отличий подходов к пониманию изоляции родились две школы юнит-тестирования: классическая и лондонская.

Классическая школа (школа Детройта или Чикаго) называется классической, потому что изначально все подходили к юнит-тестированию именно так. Подход популяризировали любители экстремального программирования, создавшие множество идей, работая над проектом Chrysler C3 в Детройте в конце 1990-х.

Лондонская школа происходит из сообщества программистов в Лондоне.

	Изоляция	Что считается юнитом (единицей кода)	Что именно изолируется
Лондонская школа	Юнитов	Класс	Любые изменяемые зависимости
Классическая школа	Юнит-тестов	Класс или набор классов	Совместные зависимости

Главное различие между школами в том, как они интерпретируют аспект изоляции в определении юнит-теста.

Лондонская школа описывает изоляцию юнит-теста как изоляцию тестируемого кода от его зависимостей. Это значит, что если класс имеет зависимость от другого класса или нескольких классов, все такие зависимости нужно заменить на тестовые зависимости.

В классической школе изолируются не фрагменты рабочего кода, а сами тесты. Такая изоляция позволяет запускать тесты параллельно, последовательно и в любом порядке, не влияя на результат их работы.

Классический подход к изоляции не запрещает тестировать несколько классов одновременно, что приводит к меньшему количеству тестовых зависимостей.

> 💡 Главное отличие школ — количество создаваемых тестовых зависимостей.
>
> В лондонской школе их много, они используются везде, где есть зависимости. В классической можно использовать реальные зависимости, поэтому тестовых не так много.
>
> Какой подход выбрать — дело программиста. Это зависит от задачи.

Использование тестовых зависимостей в модульном тестировании — спорная тема. Общая рекомендация в том, чтобы стараться использовать тестовые заглушки только там, где это действительно необходимо и поможет в тестировании.

⛔ Недостатки школ

При классическом подходе, когда всё в тестах — реальные объекты, из-за ошибки в одном из методов реального объекта, «красными» станут все тесты, в которых этот метод используется (независимо от того, тестируется он или просто используется другим объектом). Локализовать проблему будет трудно, если при написании тестов вы не думали о минимально необходимых зависимостях.

В лондонской школе, когда на заглушку заменяется всё и вся, может возникнуть другая проблема: поведенческие тесты жёстко фиксируют внутреннюю реализацию тестируемой системы (что и как вызывается, с какими аргументами, какое API используется и так далее). Это создаёт трудности при рефакторинге и возможных изменениях в SUT — вместе с кодом все тесты придётся писать заново.

✅ Преимущества школ

Классический подход обычно приводит к более качественным тестам, следовательно, лучше подходит для достижения цели юнит-тестирования — стабильного роста проекта. Причина в том, что тесты, использующие тестовые заглушки, бывают более хрупкими, чем классические, а хрупкие тесты труднее поддерживать в актуальном состоянии.

Лондонская школа отличается детализацией. Тесты высоко детализированы и проверяют только один класс за раз. Упрощается тестирование большого графа взаимосвязанных классов (когда много цикличных и сложных зависимостей). Так как все зависимости заменяются тестовыми заглушками, вам не придётся беспокоиться о них при написании теста. Если тест падает, вы точно знаете, в какой функциональности был сбой, так как все зависимости заменены на заглушки.

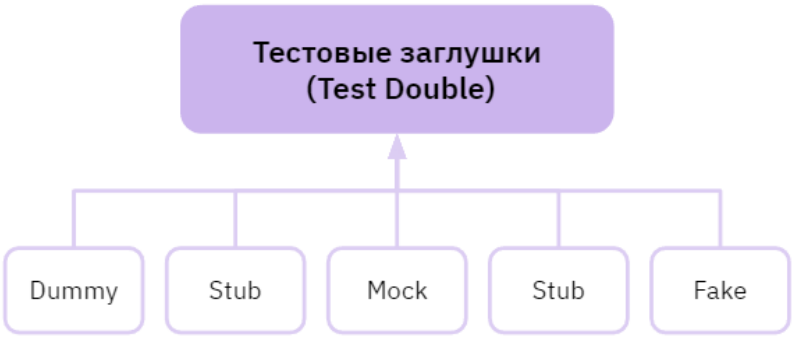
Лондонская школа	**Классическая школа**
Большое количество тестовых заглушек	В тестах чаще используются реальные объекты
⛔ При рефакторинге может понадобится заново настраивать все тестовые заглушки; ✅ Более детализированные тесты; ✅ Легче тестировать сложный (со многими зависимостями) код; ✅ Легко найти ошибку;	⛔ Возможные проблемы с поиском ошибок; ✅ **Менее хрупкие тесты;**

Типы test doubles

Часто финальная версия программного обеспечения состоит из сложного набора объектов или функций, взаимодействующих для создания конечного результата. В юнит-тестировании может потребоваться использовать объекты, которые выглядят и ведут себя как их аналоги, но на самом деле являются упрощёнными версиями, которые снижают сложность и облегчают тестирование.

Тестовая заглушка (test double) — общий термин, который описывает все разновидности фиктивных зависимостей в тестах.

Использование тестовых заглушек снижает сложность тестов и позволяет проверять код независимо от остальной системы. Джерард Месарош, автор книги «Шаблоны тестирования xUnit», перечисляет разные типы двойников:



Dummy (объект-заглушка)

Dummy (фиктивный объект или объект-заглушка) — самый простой тип заглушки. Это объект, который передаётся для удовлетворения конструктора. В самом dummy-объекте ничего не будет реализовано.

Обычно такая заглушка используется для заполнения списков параметров, просто передается тестируемой системе в качестве аргумента (или атрибута аргумента). В большинстве случаев вместо неё можно использовать значения ‘‘null’’.

Пример использования dummy-объекта:

```
public class PaymentService {

   private Logger logger;

   public PaymentService(Logger logger) {
       this.logger = logger;
   }

   public booleanPaymentRequest createPaymentRequest() {
      . . .
   }
}
```

Сам класс LoggerDummy пустой, он не содержит реализации, так как нужен только для удовлетворения конструктора:

```
public class LoggerDummy implements Logger {
// Реализация отсутствует
}
```

Из-за того, что класс PaymentService требует класс Logger в качестве обязательного параметра в конструкторе, было принято решение использовать заглушку LoggerDummy, чтобы избежать сохранения журналов логов в файл или отправки их куда-либо еще, что может нарушить изоляцию теста. Также мы не хотим проверять всю информацию, содержащуюся в журналах и не связанную с бизнес-логикой, поэтому мы создаем фиктивный объект логгера и передаем его в сервис. Это позволяет нам быть уверенными в работе наших тестов, поскольку, если тест провалится, мы можем точно знать, что проблема не в тестовом двойнике.

Тест, проверяющий часть бизнес-логики класса PaymentServiceShould:

```
class PaymentServiceShould {

   @Test
   void create_payment_request() {
        // Создается dummy-обьект без реализации
        LoggerDummy loggerDummy = new LoggerDummy();
                        . . .
        // loggerDummy передается в качестве аргумента
        PaymentService paymentService = new
        PaymentService(loggerDummy);

        assertEquals(true,
        paymentService.createPaymentRequest());
     }
}
```

Другие примеры таких dummy-объектов — new Object(), null, «Ignored String»

Stub (заглушка)

Stub (заглушки) — одни из самых популярных видов тестовых заглушек.

Stub-обьекты немного сложнее, чем dummy: они предоставляют готовые ответы на наши вызовы, в них по-прежнему нет логики, но они возвращают предопределённое значение. Это готовые ответы на вызовы, сделанные во время теста, обычно вообще не реагируя ни на что, кроме того, что запрограммировано для теста.

Такие заглушки позволяют тесту управлять опосредованным вводом тестируемой системы и провоцировать выполнение ветвей кода, которые в других условиях не выполняются.

Теперь PaymentRequest должен содержать комиссию оператора кредитной карты. Размер этой комиссии определяется какой-то определённой внутренней логикой. Чтобы реализовать это, вам нужно создать заглушку и добавить в неё необходимые изменения PaymentService.

Первый шаг — реализация интерфейса, который нужен для заглушки и производственного кода. Это та часть, которую вы разрабатываете заранее, думая о том, какими должны быть параметры в заглушке, и что должно быть возвращено. Думайте не о внутренней реализации, а о том, кто будет использовать заглушку:

```
public interface OperatorRate {
}
```

Определив интерфейс, мы можем начать писать заглушку:

```
public class OperatorRateStub implements OperatorRate {
   private int rate;
// Задается готовый ответ
   public OperatorRateStub(int rate){
       this.rate = rate;
   }
}
```

Заглушка всегда будет возвращать значение, переданное в конструкторе, у нас есть полный контроль над заглушкой, и она полностью изолирована от производственного кода. Теперь можно реализовывать тест:

```
@Test
void create_payment_request() {
        . . .
   // 10 — жестко заданное значение
   OperatorRate operatorRate = new OperatorRateStub(10);
   PaymentService paymentService = new
   PaymentService(loggerDummy, operatorRate);
        . . .
   assertEquals(true, paymentService.createPaymentRequest());
}
```

Mock (имитация)

Mock (имитация) помогает эмулировать и проверять выходные взаимодействия — то есть вызовы, совершаемые тестируемой системой к её зависимостям для изменения их состояния. Они могут выдавать исключение, если получают вызов, которого не ожидают.

Общий принцип работы с моками:

1. Настраиваем ожидания.
2. Вызываем метод тестируемой системы.
3. Проверяем, был ли метод вызван в конце.

В качестве примера продолжим развивать историю с финансовой системой: теперь нам нужно добавить функционал отправки электронного письма. Есть две причины для изоляции метода отправки электронной почты:

1. Отправка электронных писем — это деятельность, которая общается с внешним миром. Мы не можем отправлять электронное письмо каждый раз, когда запускаем наши тесты, — это их замедлит.
2. PaymentService не нужно знать о реализации отправления электронной почты, так как смешивание этих двух бизнес-процессов создаст лишнюю связь и усложнит обслуживание сервиса или рефакторинга способа отправки электронных писем.

Шаги, которым мы должны следовать:

1. Создание интерфейса.
2. Создание мока, реализующего интерфейс.
3. Написание теста.

Интерфейс:

```
public interface PaymentEmailSender {
    void send(PaymentRequest paymentRequest);
}
```

Затем мы должны реализовать наш мок:

```
public class PaymentServiceMock implements PaymentEmailSender {

     . . .
```

```
    public void expect(PaymentRequest paymentRequest) {
        expectedPaymentRequest.add(paymentRequest);
    }

    public void verify() {
        assertEquals(paymentRequestSent, expectedPaymentRequest);
    }
}
```

Добавляем два метода для настройки моки: expect и verify. Verify использует assertEqual метод JUnit для сравнения ожидаемого значения с тем, которое передаётся тестируемой системой. Таким образом мы можем проверить выходные значения.

Пишем тест для новой функции:

```
@Test
void send_email_to_the_administration_if_sale_is_over_1000() {
    EmailSenderMock emailSender = new EmailSenderMock();
      . . .
PaymentService paymentService = new PaymentService(loggerDummy,
operatorRate, emailSender);

    paymentService.createPaymentRequest();
    // проверяются выходные взаимодействия
    emailSender.expect(paymentRequest);
    emailSender.verify();
}
```

Spy (шпионы)

Spy (шпионы) — это заглушки, которые также записывают некоторую информацию, основанную на том, как они были вызваны.

Test spy (тестовый шпион) используется для тестов взаимодействия. Основная функция — запись данных и вызовов, поступающих из тестируемого объекта, для последующей проверки корректности вызова зависимого объекта. Он позволяет проверить логику именно нашего тестируемого объекта без проверок зависимых объектов.

> 💡 Думайте о шпионе как о ком-то, кто внедрился в вашу тестируемую систему и записывает каждое её движение, — совсем как киношный шпион.
>
> В отличие от mocks, шпион молчит. Вам решать, на основе каких данных он должен вызываться.

Шпионы используются в тестировании, когда мы не можем быть уверены, что наша система вызовет нужный метод зависимости. Мы создаём шпиона для этой зависимости и записываем все вызовы, чтобы убедиться, что тестируемая система вызывает нужные данные. Если мы не записываем нужные вызовы, то мы можем использовать утверждения, чтобы проверить, что эти вызовы произошли, и что наша система работает правильно.

Для примера можем использовать тот же интерфейс, который создали для mock, но реализуем новый тест с помощью шпиона.

```
public class PaymentEmailSpy implements PaymentEmailSender {

    private List<PaymentRequest> paymentRequests = new ArrayList<>();

    @Override
    public void send(PaymentRequest paymentRequest) {
        paymentRequests.add(paymentRequest);
    }

    public int timesCalled() {
        return paymentRequests.size();
    }

    public boolean calledWith(PaymentRequest paymentRequest) {
        return paymentRequests.contains(paymentRequest);
    }
}
```

Реализация Spy близка к макету, но вместо того, чтобы выдавать вызовы, которые мы ожидаем, мы просто записываем поведение класса, затем переходим к тестированию и можем утверждать, что нам там нужно.

```
class PaymentServiceShould {

            . . .

    @Test
    void not_send_email_for_sales_under_1000() {
```

```
            . . .

        EmailSenderSpy emailSpy = new EmailSenderSpy();
        PaymentService spiedPaymentService = new
        PaymentService(loggerDummy, operatorRate, emailSpy);

        spiedPaymentService.createPaymentRequest();

        assertEquals(0, emailSpy.timesCalled());
    }
}
```

Fake (подделки)

Fake (подделка) — это тестовая заглушка, задача которой очень похожа на стаб: предоставить простые и быстрые ответы клиенту, который его потребляет. Основное отличие в том, что фейк использует простую и легковесную рабочую реализацию под капотом.

> 💡 Fake используется, чтобы запускать тесты быстрее. Это замена тяжеловесного внешнего зависимого объекта легковесной реализацией. Примеры — эмулятор для конкретного приложения БД в памяти (fake database) или фальшивый веб-сервис.

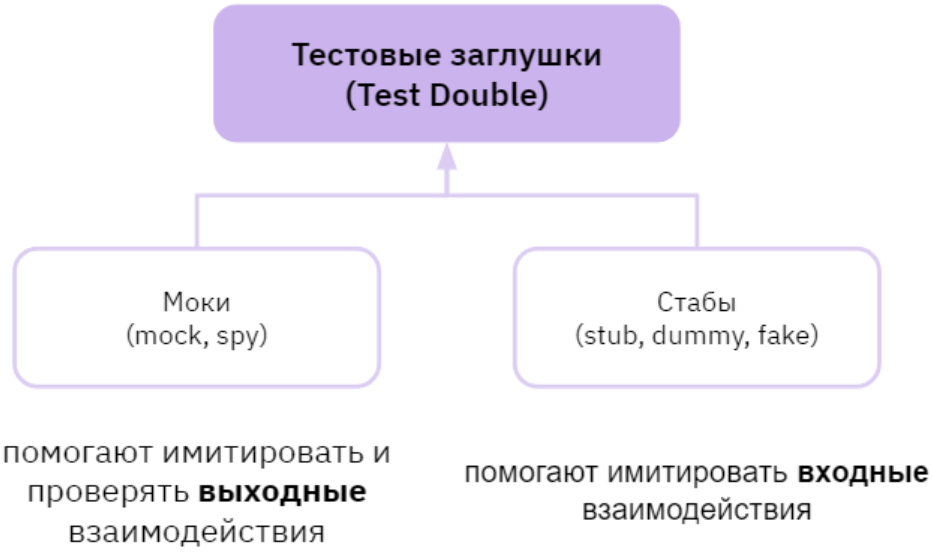
Примером может быть реализация объекта доступа к данным или репозитория в памяти. Эта поддельная реализация сервиса авторизации не будет задействовать базу данных, но будет использовать простую коллекцию для хранения данных. Это позволяет нам проводить интеграционное тестирование сервисов без запуска базы данных и выполнения трудоёмких запросов:

```
public class FakeAccountRepository implements AccountRepository {

   Map<User, Account> accounts = new HashMap<>();

   public FakeAccountRepository() {
        this.accounts.put(new User("john@mail.com"), new
        UserAccount());
        this.accounts.put(new User("boby@mail.com"), new
        AdminAccount());
   }

   String getPasswordHash(User user) {
        return accounts.get(user).getPasswordHash();
```

```
    }
}
```

💡 Такое разнообразие может показаться слишком сложным, но на деле все разновидности можно разделить на два типа: моки и стабы. Их чаще всего используют на практике, а остальные типы заглушек можно рассматривать как их подтипы, так как различия между ними незначительны.

Далее, на практике, будем использовать упрощённую классификацию, так как все различия между тестовыми заглушками сводятся к следующему:



🔥 **Моки помогают эмулировать и проверять выходные взаимодействия** — то есть вызовы, совершаемые тестируемой системой к её зависимостям для изменения их состояния.

🔥 **Стабы помогают эмулировать входные взаимодействия** — то есть вызовы, совершаемые тестируемой системой к её зависимостям для получения входных данных.

Все остальные различия между пятью разновидностями незначительны. Например, шпионы выполняют ту же функцию, что и моки, а отличаются от них лишь тем, что пишутся вручную, тогда как моки создаются при помощи мок-фреймворков.

Mocking Frameworks

Установка Mockito

Mocking — это процесс, используемый в модульном тестировании, когда внешние зависимости тестируемого модуля заменяются на тестовые заглушки.

Цель mocking — изолировать и сосредоточиться на тестируемом коде, а не на поведении или состоянии внешних зависимостей.

В mocking зависимости заменяются объектами с тщательно контролируемыми заменами, которые имитируют поведение реальных объектов.

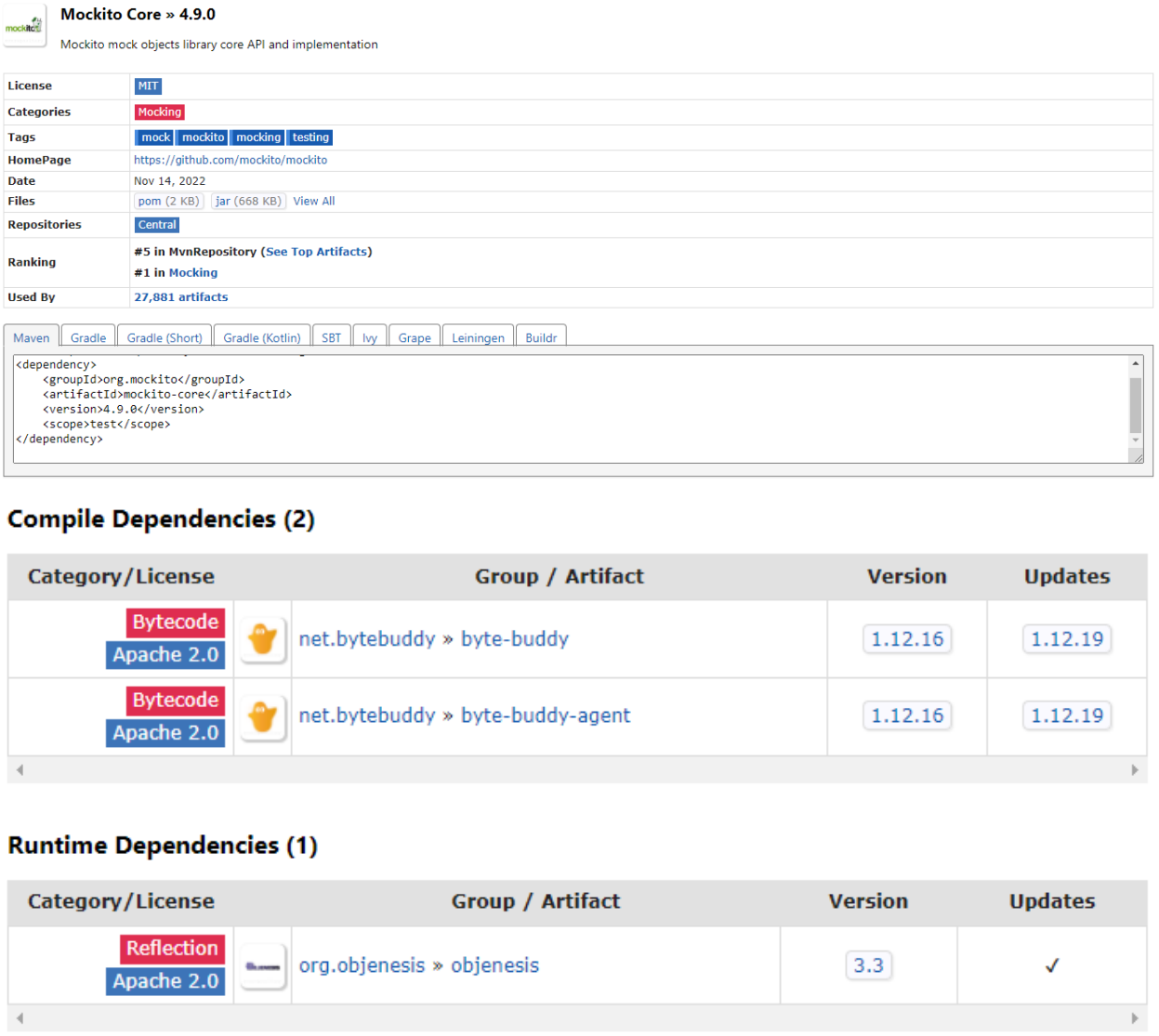
Mockito — это фреймворк с открытым исходным кодом, который позволяет нам легко создавать тестовые заглушки. В Mockito мы можем работать со следующими типами заглушек:

- Stubs (стабы)
- Spies (шпионы)
- Mocks (макеты)

В Mockito мы можем заменять заглушками как интерфейсы, так и надклассы. Фреймворк также помогает минимизировать шаблонный код при использовании аннотаций.

Установим фреймворк в проект. Для этого можем воспользоваться либо инструментами сборки, либо установить зависимости вручную, как мы и сделаем.

Переходим в репозиторий **maven-central** и скачиваем jar-архив **mockito-core** с последней версией из общедоступного репозитория.



Mockito Core » 4.9.0

Mockito mock objects library core API and implementation

License	MIT
Categories	Mocking
Tags	mock mockito mocking testing
HomePage	https://github.com/mockito/mockito
Date	Nov 14, 2022
Files	pom (2 KB) jar (668 KB) View All
Repositories	Central
Ranking	#5 in MvnRepository (See Top Artifacts) #1 in Mocking
Used By	27,881 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.9.0</version>
    <scope>test</scope>
</dependency>
```

Compile Dependencies (2)

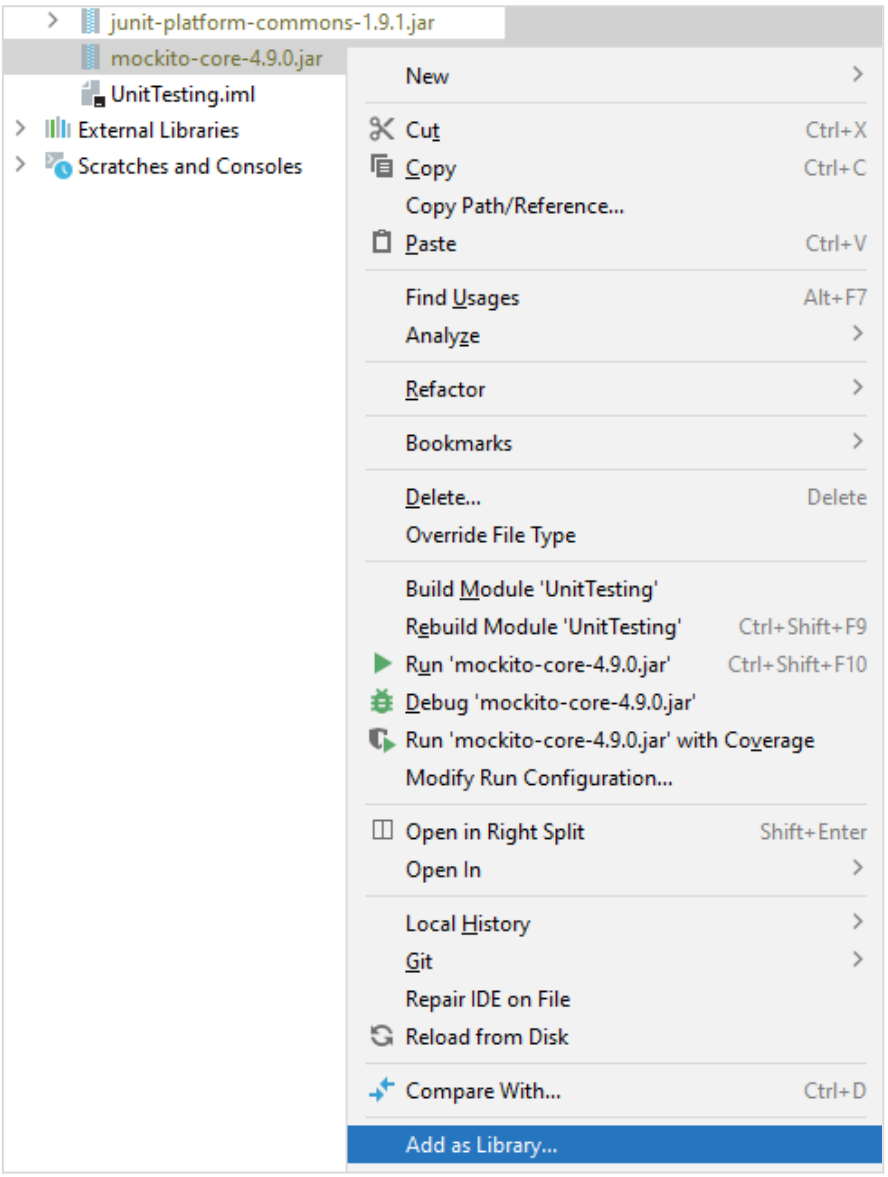
Category/License	Group / Artifact	Version	Updates
Bytecode Apache 2.0	net.bytebuddy » byte-buddy	1.12.16	1.12.19
Bytecode Apache 2.0	net.bytebuddy » byte-buddy-agent	1.12.16	1.12.19

Runtime Dependencies (1)

Category/License	Group / Artifact	Version	Updates
Reflection Apache 2.0	org.objenesis » objenesis	3.3	✓

Для использования фреймворка Mockito необходимо установить две дополнительные библиотеки, которые совместимы с основной версией фреймворка. В мавен-репозитории ссылки на эти библиотеки находятся на той же странице, где расположен фреймворк. Это библиотеки:

- **Byte Buddy** — Java-библиотека для создания Java-классов во время выполнения.
- **Objenesis** — библиотека для создания экземпляров объектов Java.



Проверим, что фреймворк работает — создадим мок-объект списка:

```
import java.util.List;

import static org.mockito.Mockito.*;

public class Main {
   public static void main(String[] args) {
        // Создание mock
        List mockedList = mock(List.class);

        // Использование
        mockedList.add("one");
        mockedList.clear();
    }
}
```

Использование Mockito

Mockito позволяет создавать mock-объекты разными методами.

1. С помощью статического метода mock().

```
List mockedList = mock(List.class);
```

2. С использованием аннотации @Mock.

```
@Mock
private Calculator calculator;
```

Вы можете использовать конструкцию **when(...).thenReturn(...)** для указания условия и возвращаемого значения для этого условия:

```
LinkedList mockedLinkedList = mock(LinkedList.class);

when(mockedLinkedList.get(0)).thenReturn("nullValue");

// Выведет "nullValue"
System.out.println(mockedLinkedList.get(0));
```

Можно настраивать выбрасываемое исключение:

```
LinkedList mockedLinkedList = mock(LinkedList.class);

when(mockedLinkedList.get(1)).thenThrow(new RuntimeException());

// Вернет исключение runtime exception
System.out.println(mockedLinkedList.get(1));
```

Mockito отслеживает все вызовы методов и их параметров для mock-объекта. Вы можете использовать метод verify(), чтобы убедиться, что метод был вызван с определёнными параметрами.

```
LinkedList mockedLinkedList = mock(LinkedList.class);

when(mockedLinkedList.get(0)).thenReturn("nullValue");
when(mockedLinkedList.get(1)).thenThrow(new RuntimeException());
// Выведет "nullValue"
System.out.println(mockedLinkedList.get(0));

// Вернет исключение runtime exception
System.out.println(mockedLinkedList.get(1));
```

```
// Если mockedLinkedList.get(0) не будет вызван до этой проверки,
то тест провалится
verify(mockedLinkedList).get(0);
```

Вызов реальных методов:

```
when(mock.someMethod()).thenCallRealMethod()
```

Проверка точного количества вызовов:

```
verify(mockedList, times(2)).add("twice");
verify(mockedList, never()).add("never happened");
```

Проверка с таймаутом:

```
verify(mock, timeout(100)).someMethod();
```

Есть также класс BDDMockito, предоставляющий примерно те же возможности что и класс Mockito в форме, более подходящей для BDD.

```
import static org.mockito.BDDMockito.*;

 Seller seller = mock(Seller.class);
 Shop shop = new Shop(seller);

 public void shouldBuyBread() throws Exception {
   //given
   given(seller.askForBread()).willReturn(new Bread());

   //when
   Goods goods = shop.buyBread();

   //then
   assertThat(goods, containBread());
 }
```

Другие mocking-фреймворки

EasyMock

Фреймворк EasyMock создаёт мок-объекты, используя объект java.lang.reflect.Proxy. Когда мы создаём мок, во время выполнения теста прокси-объект заменяет

реальный объект. Прокси-объект получает свои поля и методы из интерфейса или класса, которые мы передаём при создании макета.

Создание теста с EasyMock состоит из трёх этапов (модель Record-Replay-Verify):

1. **Создание** моков, поведение которых мы хотим делегировать прокси-объектам. Как правило, мы имитируем классы, которые взаимодействуют с внешними системами или классами, которые не должны быть частью тестового кода.
2. **Запись** — ожидание от моков. Включают в себя моделирование метода с определёнными аргументами, возвращаемое значение вызываемого метода и количество раз, когда метод должен быть вызван.
3. **Повтор** — делает макет объекта доступным. В режиме повтора, когда тест вызывает записанный метод, макет возвращает записанные результаты на предыдущем шаге.

WireMock

Архитектура микросервисов позволяет нам разрабатывать, тестировать и развёртывать различные компоненты приложения независимо. Хотя такой компонент может быть разработан независимо, тестирование его в изоляции может быть сложной задачей.

Для настоящего интеграционного тестирования микросервиса мы должны протестировать его взаимодействие с другими API. WireMock помогает в интеграционном тестировании, когда нам нужно имитировать внешние API, для тестирования определенного API, зависящего от этих внешних API, для завершения транзакции. WireMock — это популярный HTTP-макет сервера, который помогает в моделировании зависимостей API и HTTP-ответов.

MockWebServer

MockWebServer — полезная библиотека для моделирования зависимых API, от которых зависит текущий компонент (в процессе тестирования). Такие мок API чрезвычайно полезны в архитектуре микросервисов, где мы разрабатываем несколько зависимых сервисов одновременно. MockWebServer чем-то похож на WireMock, и его можно использовать для тестирования асинхронных HTTP-вызовов, выполняемых из Spring WebClient.

JMockit

JMockit — это открытое программное обеспечение, которое предоставляет поддержку для создания макетов, подделок, интеграционного тестирования и инструментов для анализа покрытия кода. Это решение используется для замещения внешних зависимостей вне границ тестирования, подобно Mockito. Однако у JMockit есть важная особенность — он позволяет имитировать любые объекты, включая те, которые сложно подделать с помощью других библиотек, таких как приватные методы, конструкторы, статические и финальные методы.

Как и в EasyMock, в JMockit используется модель «Record-Replay-Verify» в тестировании, но JMockit позволяет определять ожидаемые результаты и проверки очень подробно и декларативно. Для использования JMockit необходимо скачать и установить соответствующие библиотеки, которые можно найти в мавен-репозитории на странице с фреймворком.

PowerMock

PowerMock расширяет существующие мок-фреймворки (EasyMock и Mockito), добавляя к ним ещё более мощные функции. PowerMock позволяет писать хорошие модульные тесты даже для самого непроверяемого кода.

Например, большинство макетных фреймворков в Java не могут имитировать статические методы или конечные классы. Но используя PowerMock, мы можем издеваться практически над любым классом.

Сейчас PowerMock расширяет фреймворки для редактирования EasyMock и Mockito. В зависимости от того, какое расширение предпочтительнее, синтаксис для написания любого модульного теста немного отличается.

Подведём итоги

На этом уроке мы:

- Узнали, что называют зависимостью и какие виды зависимостей бывают, как они помогают или мешают тестировать код, какие подходы существуют на практике.
- Познакомились с тестовыми заглушками и разобрались, когда применяются разные виды заглушек.
- Рассмотрели фреймворки, которые помогают нам работать с зависимостями напрямую.

Домашнее задание

Установите фреймворк и дополнительные зависимости, проверьте, что основной класс Mockito импортируется.

Что можно почитать ещё?

1. Test double — Wikipedia
2. Mockito и как его готовить
3. Mockito
4. Introduction to Test Doubles — Java Code Geeks — 2023
5. TestDouble
6. Test Double at XUnitPatterns.com
7. Introduction to Test Doubles — Java Code Geeks — 2023
8. Mockito 2.2.7 API
9. Mocks Aren't Stubs
10. Introduction to Test Doubles — Java Code Geeks — 2023
11. BDDMockito (Mockito 5.0.0 API)

Используемая литература

1. Хориков Владимир «Принципы юнит-тестирования». — СПб.: Питер, 2021. — 320 с.
2. Джерард Месарош «Шаблоны тестирования xUnit. Рефакторинг кода тестов» – Вильямс, 2016 г. – 832 с