# Day 06 - Piscine Java

JUnit/Mockito

*Резюме: Сегодня вы познакомитесь с основами модульного и интеграционного тестирования*

# Contents

# Chapter I
# Preamble

Модульные и интеграционные тесты позволяют программисту убедиться в правильности работы написанных им программ. Данные методы тестирования выполняются в автоматическом режиме.

Таким образом, вашей задачей является написание не только правильного кода, но и кода, способного выполнить проверку вашей реализации.

Модульные тесты в Java - классы, содержащие несколько тест-методов для публичных методов тестируемых классов. Каждый класс модульных тестов должен проверять функциональность ровно одного класса. Такое тестирование позволяет достоверно выявлять места возникновения ошибок. Для того, чтобы выполнять тестирование без конкретных зависимостей, используются объекты-заглушки с временной реализацией.

В противовес модульным тестам, интеграционные тесты позволяют проверять связки различных компонентов.

Ниже приведено несколько лучших практик модульного и интеграционного тестирования:

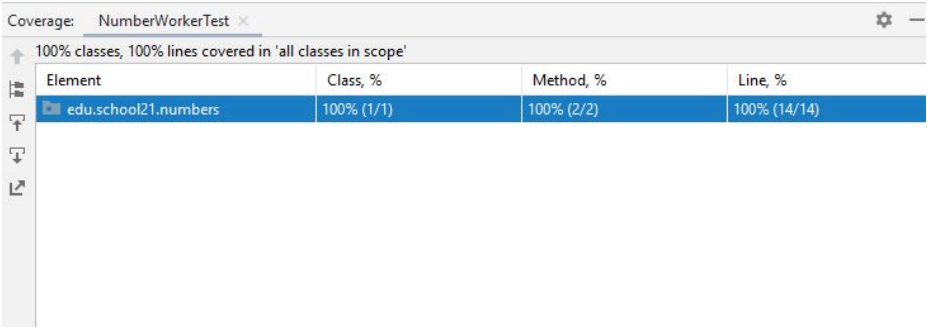
1. Используйте адекватные имена для тест-методов.
2. Рассматривайте различные ситуации.
3. Обеспечивайте минимум 80-процентное покрытие кода тестами.
4. Каждый тест-метод должен содержать небольшое количество кода и выполняться быстро.
5. Тест-методы должны быть изолированы друг от друга и не иметь побочных эффектов.

# Chapter II
## General Rules

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- Сейчас для вас существует только одна версия Java - 1.8. Убедитесь, что на вашем компьютере установлен компилятор и интерпретатор данной версии.
- Не запрещено использовать IDE для написания исходного кода и его отладки.
- Код чаще читается, чем пишется. Внимательно изучите представленный документ с правилами оформления кода. В каждом задании обязательно придерживайтесь общепринятых стандартов Oracle - https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
- Комментарии в исходном коде вашего решения запрещены. Они мешают восприятию.
- Pay attention to the permissions of your files and directories.
- To be assessed your solution must be in your GIT repository.
- Your solutions will be evaluated by your piscine mates.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google. И еще, для любых ваших вопросов существует ответ на Stackoverflow. Научитесь правильно их задавать.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- And may the Force be with you!
- Не откладывайте на завтра то, что можно было сделать вчера ;)

# Chapter III
# Rules of the day

- Во всех заданиях используйте фреймворк JUnit 5
- Используйте следующие зависимости и плагины для корректной работы:
    - `maven-surefire-plugin`
    - `junit-jupiter-engine`
    - `junit-jupiter-params`
    - `junit-jupiter-api`
- Все тесты должны быть доступны для запуска через выполнение команды `mvn clean compile test`
- Для всех реализованных тестов необходимо 100% покрытие исходного кода тестируемого класса. Ниже приведен пример демонстрации полного покрытия средствами IntelliJ IDEA для Exercise 00:



# Chapter IV
## Exercise 00 - First Tests

| Exercise 00: First Tests | |
|---|---|
| Turn-in directory | ex00 |
| Files to turn-in | Tests-folder |

Сейчас необходимо реализовать класс `NumberWorker`, который содержит следующую функциональность:

```
public boolean isPrime(int number) {
...
}
```

Данный метод определяет, является ли число простым и возвращает `true/false` для всех натуральных (целых положительных) чисел. Для отрицательных чисел, а также для 0 и 1 программа должна выбросить непроверяемое исключение `IllegalNumberException`.

```
public int digitsSum(int number) {
...
}
```

Данный метод возвращает сумму цифр исходного числа.

Также необходимо создать класс `NumberWorkerTest`, реализующий логику модульного тестирования. Методы класса `NumberWorkerTest` должны проверить корректность работы методов `NumberWorker` на различных входных данных:

1. метод `isPrimeForPrimes` для проверки `isPrime` на простых числах (не менее 3-х)
2. метод `isPrimeForNotPrimes` для проверки `isPrime` на составных числах (не менее 3-х)
3. метод `isPrimeForIncorrectNumbers` для проверки `isPrime` на некорректных числах (не менее 3-х)
4. метод проверки `digitsSum` на наборе, состоящем не менее чем из 10-ти чисел
5. метод проверки `digitsSum` на некорректном наборе.

**Требования:**

- В классе `NumberWorkerTest` должно быть не более 4-х методов, тестирующих функциональность `NumberWorker`
- Для методов 1-3 обязательно использование `@ParameterizedTest` и `@ValueSource`
- Для метода 4 обязательно использование `@ParameterizedTest` и `@CsvFileSource`
- Для метода 4 необходимо подготовить файл `data.csv`, в котором вы укажете не менее 10 целых чисел и корректную сумму их цифр. Пример содержимого файла: 1234, 10

- Аналогично для метода 5 необходимо подготовить файл `data_fails.csv` с набором некорректных чисел.

**Структура проекта:**

- `Tests`
  - `src`
    - `main`
      - `java`
        - `edu.school21.numbers`
          - `NumberWorker`
      - `resources`
    - `test`
      - `java`
        - `edu.school21.numbers`
          - `NumberWorkerTest`
      - `resources`
        - `data.csv`
        - `data_fails.csv`
  - `pom.xml`

# Chapter V
## Exercise 01 - Embedded DataBase

| Exercise 01: Embedded DataBase | |
|---|---|
| Turn-in directory | ex01 |
| Files to turn-in | Tests |

Для реализации интеграционного тестирования компонентов, взаимодействующих с базой данных, не следует использовать тяжеловесную СУБД типа PostgreSQL. Лучшей практикой является создание легкой in-memory базы данных с заранее подготовленными данными.

Реализуйте механизм создания DataSource для СУБД HSQL. Для этого подключите к проекту зависимости `spring-jdbc` и `hsqldb`. Подготовьте файлы `schema.sql` и `data.sql`, в котором вы опишете структуру таблицы `product` и тестовые данные (не менее 5-ти).

Структура таблицы `Product`:

- идентификатор
- наименование
- стоимость

Также создайте класс `EmbeddedDataSourceTest`. В данном классе реализуйте метод init(), помеченный аннотацией `@BeforeEach`. В данном классе реализуйте функционал по созданию DataSource с использованием `EmbeddedDataBaseBuilder` (класс библиотеки `spring-jdbc`). Реализуйте простой тестовый метод, проверяющий возвращаемое значение метода `getConnection()`, созданного `DataSource` (данное значение не должно быть `null`).

**Структура проекта:**

- Tests
    - src
        - main
            - java
                - edu.school21.numbers
                    - NumberWorker
            - resources
        - test
            - java
                - edu.school21
                    - numbers
                        - NumberWorkerTest
                    - repositories
                        - EmbeddedDataSourceTest
            - resources
                - data.csv
                - data_fails.csv
                - schema.sql
                - data.sql
    - pom.xml

# Chapter VI
## Exercise 02 - Test For JDBC Repository

| Exercise 02: Test For JDBC Repository | |
|---|---|
| Turn-in directory | ex02 |
| Files to turn-in | Tests |

Реализуйте пару интерфейс-класс `ProductsRepository/ProductsRepositoryJdbcImpl` со следующими методами:

```
List<Product> findAll()

Optional<Product> findById(Long id)

void update(Product product)

void save(Product product)

void delete(Long id)
```

Необходимо реализовать класс `ProductsRepositoryJdbcImplTest`, методы которого должны проверять функциональность репозитория с использованием in-memory базы данных из предыдущего упражнения. В данном классе необходимо заранее подготовить объекты моделей, сравнение с которыми будет происходить во всех тестах.

Пример объявления тестовых данных представлен ниже:

```
class ProductsRepositoryJdbcImplTest {

       final List<Product> EXPECTED_FIND_ALL_PRODUCTS = ...;

       final Product EXPECTED_FIND_BY_ID_PRODUCT = ...;

       final Product EXPECTED_UPDATED_PRODUCT = ...;

}
```

**Примечания:**

1. Каждый тест должен быть изолирован от поведения других тестов. Следовательно, перед выполнением каждого теста база данных должна быть в исходном состоянии.
2. Тестовые методы могут использовать вызовы других методов, не тестируемых в данном. Например, тест метода `update()` может вызвать метод `findById()` для проверки успешности обновления сущности в базе данных.

**Структура проекта:**

- Tests
  - src
    - main
      - java
        - edu.school21
          - numbers
            - NumberWorker
          - models
            - Product
          - repositories
            - ProductsRepository
            - ProductsReposutoryJdbcImpl
      - resources
    - test
      - java
        - edu.school21
          - numbers
            - NumberWorkerTest
          - repositories
            - EmbeddedDataSourceTest
            - ProductsReposutoryJdbcImplTest
      - resources
        - data.csv
        - data_fails.csv
        - schema.sql
        - data.sql
  - pom.xml

# Chapter VII
## Exercise 03 - Test for Service

| Exercise 03: Test for Service | |
|---|---|
| Turn-in directory | ex03 |
| Files to turn-in | Tests |

Важное правило модульных тестов - тестирование отдельного компонента системы без обращения к функциональности его зависимостей. Такой подход позволит разработчикам создавать и тестировать компоненты независимо друг от друга, а также отложить реализацию каких-либо конкретных частей приложения.

Сейчас вам необходимо реализовать слой бизнес-логики, представленный классом `UsersServiceImpl`. Данный класс содержит логику аутентификации пользователя. Помимо этого, класс имеет зависимость на интерфейс `UsersRepository` (в данном задании не требуется создавать имплементацию данного интерфейса).

Описанный вами интерфейс `UsersRepository` должен содержать следующие методы:

```
User findByLogin(String login);

void update(User user);
```

Предполагается, что метод `findByLogin` возвращает найденный по логину объект `User`, либо выбрасывает исключение `EntityNotFoundException`, если пользователь с указанным логином не был найден. Метод update выбрасывает аналогичное исключение при обновлении не существующего в базе данных пользователя.

Сущность `User` должна содержать следующие поля:

- Идентификатор
- Логин
- Пароль
- Статус успешности авторизации (`true` - аутентифицирован, `false` - не аутентифицирован)

В свою очередь, класс `UsersServiceImpl` вызывает данные методы внутри функции аутентификации:

```
boolean authenticate(String login, String password)
```

Данный метод:

1. Проверяет, не был ли пользователь с данным логином уже аутентифицирован в системе. В случае, если аутентификация была проведена необходимо выбросить исключение `AlreadyAuthenticatedException`.
2. Получает пользователя с данным логином из `UsersRepository`
3. Если пароль полученного пользователя совпадает с указанным - метод задает статус успешности аутентификации для данного пользователя, обновляет

информацию о нем в БД, а также возвращает `true`. В случае несовпадения пароля метод возвращает `false`.

Таким образом, вашей задачей является:

1. Создание интерфейса `UsersRepository`
2. Создание класса `UsersServiceImpl` и метода `authenticate`
3. Написание модульного теста для класса `UsersServiceImpl`

Поскольку ваша цель проверить корректность работы метода `authenticate` независимо от компонента `UsersRepository`, вам необходимо использовать `mock`-объект и заглушки методов `findById` и `update` (см. библиотеку `Mockito`).

Метод `authenticate` должен быть проверен для трех случаев:

1. Для корректного логина-пароля (проверить вызов метода `update` с помощью `verify`-инструкции библиотеки Mockito)
2. Для некорректного логина
3. Для некорректного пароля
4. Для уже аутентифицированного пользователя

**Структура проекта:**

- Tests
  - src
    - main
      - java
        - edu.school21
          - exceptions
            - AlreadyAuthenticatedException
          - numbers
            - NumberWorker
          - models
            - Product
            - User
          - services
            - UsersServiceImpl
          - repositories
            - ProductsRepository
            - ProductsReposutoryJdbcImpl
            - UsersRepository
      - resources
    - test
      - java
        - edu.school21
          - services
            - UsersServiceImplTest
          - numbers
            - NumberWorkerTest
          - repositories
            - EmbeddedDataSourceTest
            - ProductsReposutoryJdbcImplTest
      - resources
        - data.csv
        - data_fails.csv
        - schema.sql
        - data.sql
  - pom.xml

**CHECKLIST**

https://docs.google.com/document/d/1U4sK-h9mUVNGHyuxDwacIRXWX6k_a5tWk80Zj8KxNjg/edit?usp=sharing