

## Модуль 06 - Бассейн Java

JUnit/Mockito

Резюме: Сегодня вы изучите основы модульного и интеграционного тестирования.

Содержание

I Предисловие

II Инструкции

III Правила дня

IV Упражнение 00: Первые тесты

V Упражнение 01. Встроенная база данных

VI Упражнение 02. Проверка репозитория JDBC

VII Упражнение 03: Тест на служение

# Глава I

## Предисловие

Модульные и интеграционные тесты позволяют программисту убедиться в правильности работы создаваемых им программ. Эти методы тестирования выполняются автоматически.

Таким образом, ваша цель — не только написать правильный код, но и создать код для проверки правильности вашей реализации.

Модульные тесты в Java — это классы, содержащие несколько методов тестирования общедоступных методов тестируемых классов. Каждый класс тестирования модуля должен проверять функциональность только одного класса. Такие тесты позволяют точно выявлять ошибки. Для выполнения тестов без конкретных зависимостей используются объекты-заглушки с временной реализацией.

В отличие от модульных тестов, интеграционные тесты позволяют проверять пакеты различных компонентов.

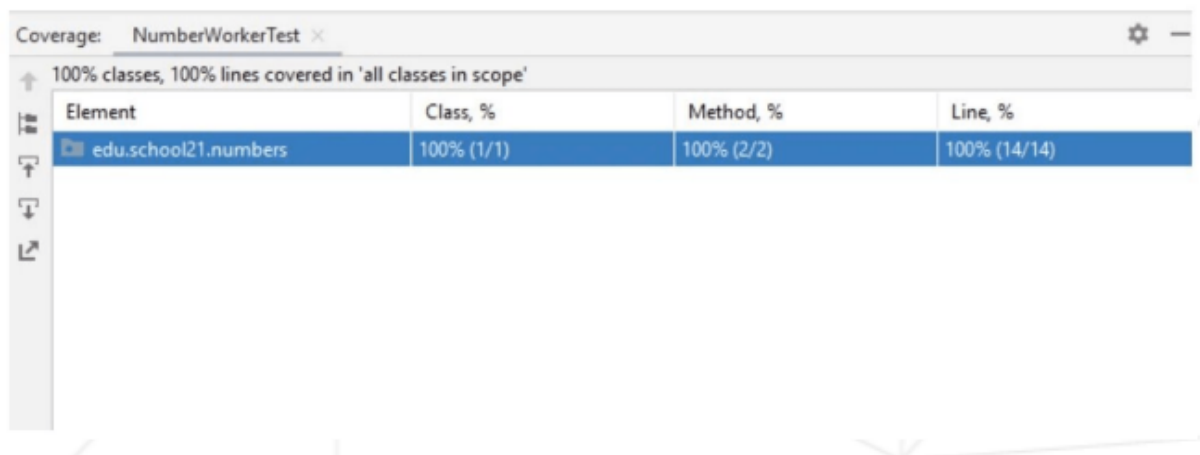
Ниже приведены несколько рекомендаций по модульному и интеграционному тестированию:

- Используйте адекватные имена для методов тестирования.
- Рассмотрите различные ситуации.
- Убедитесь, что тесты охватывают не менее 80 % кода.
- Каждый метод тестирования должен содержать небольшую часть кода и выполняться быстро.
- Методы тестирования должны быть изолированы друг от друга и не иметь побочных эффектов.

## Глава III

### Правила дня

- Используйте платформу JUnit 5 во всех задачах.
- Используйте следующие зависимости и плагины для обеспечения корректной работы:
  - плагин maven-surefire
  - junit-jupiter-engine
  - junit-jupiter-params
  - junit-jupiter-api
- Все тесты должны запускаться с помощью команды `mvn clean compile test`.
- Исходный код тестируемого класса должен быть полностью охвачен всеми реализованными тестами. Ниже приведен пример демонстрации полного охвата с помощью IntelliJ IDEA для упражнения 00:




The screenshot shows the IntelliJ IDEA Coverage tool window. The title bar reads "Coverage: NumberWorkerTest". Below the title bar, a summary bar states "100% classes, 100% lines covered in 'all classes in scope'". A table displays the coverage details for the class "edu.school21.numbers". The table has four columns: "Element", "Class, %", "Method, %", and "Line, %". The row for "edu.school21.numbers" shows 100% (1/1) for the class, 100% (2/2) for the methods, and 100% (14/14) for the lines. The table is highlighted in blue.

Element	Class, %	Method, %	Line, %
edu.school21.numbers	100% (1/1)	100% (2/2)	100% (14/14)

## Глава IV

### Упражнение 00: Первые тесты

	Exercise 00
First Tests	
Turn-in directory : ex00/	
Files to turn in : Tests-folder	
Allowed functions : All	

Теперь вам нужно реализовать класс `NumberWorker`, который содержит следующий функционал:

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

Этот метод определяет, является ли число простым, и возвращает значение `true/false` для всех натуральных (положительных целых) чисел. Для отрицательных чисел, а также 0 и 1 программа выдает непроверенное исключение. `IllegalNumberException`

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

Этот метод возвращает сумму цифр исходного номера.

Также нам необходимо создать класс `NumberWorkerTest`, реализующий логику тестирования модуля.

Методы класса `NumberWorkerTest` должны проверять правильность работы методов `NumberWorker` для различных входных данных:

1. Метод `isPrimeForPrimes` для проверки `isPrime` по простым числам (не менее трех)
2. метод `isPrimeForNotPrimes` для проверки `isPrime` по составным числам (не менее трех)
3. Метод `isPrimeForIncorrectNumbers` для проверки `isPrime` с использованием неправильных чисел (не менее трех)
4. метод проверки `digitSum` с использованием набора не менее 10 чисел

Требования:

- Класс `NumberWorkerTest` должен содержать не менее 4 методов для проверки функциональности `NumberWorker`.
- Использование `@ParameterizedTest` и `@ValueSource` обязательно для методов 1-3.
- Использование `@ParameterizedTest` и `@CsvFileSource` является обязательным для метода 4.
- Вам необходимо подготовить файл `data.csv` для метода 4, где вы должны указать не менее 10 чисел и их правильную сумму цифр. Пример содержимого файла:
  - 1234, 10

Структура проекта:


Tests

- src
  - main
    - java
      - edu.school21.numbers
        - NumberWorker
    - resources
  - test
    - java
      - edu.school21.numbers
        - NumberWorkerTest
    - resources
      - data.csv

pom.xml

## Глава V

### Упражнение 01. Встроенная база данных

	Exercise 01
Embedded DataBase	
Turn-in directory : <i>ex01/</i>	
Files to turn in : Tests	
Allowed functions : All	

Не используйте тяжелую СУБД (мунa PostgreSQL) для реализации интеграционного тестирования компонентов, взаимодействующих с базой данных.

Лучше всего создать легковесную базу данных в памяти с заранее подготовленными данными.

Реализовать механизм создания DataSource для СУБД HSQL.

Для этого подключите к проекту зависимости springjdbc и hsqldb. Подготовьте файлы `schema.sql` и `data.sql`, в которых вы будете описывать структуру таблицы продуктов и тестовые данные (не менее пяти).

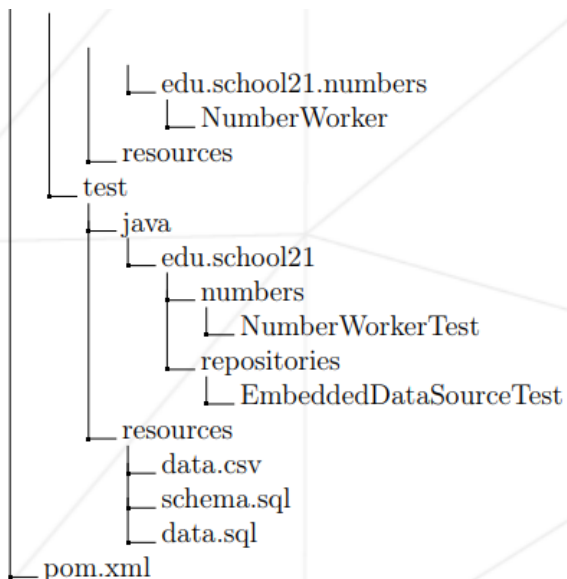
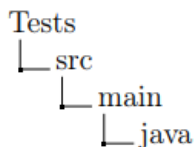
Структура таблицы продуктов:

- identifier
- name
- price

Также создайте класс `EmbeddedDataSourceTest`. В этом классе реализуйте метод `init()`, помеченный аннотацией `@BeforeEach`.


В этом классе реализуйте функциональность для создания DataSource с помощью `EmbeddedDataBaseBuilder` (класс в библиотеке `spring-jdbc`). Реализуйте простой тестовый метод для проверки возвращаемого значения метода `getConnection()`, созданного DataSource (это значение не должно быть нулевым).

Структура проекта:



## Глава VI

### Упражнение 02. Проверка репозитория JDBC

	Exercise 02
Test for JDBC Repository	
Turn-in directory : ex02/	
Files to turn in : Tests	
Allowed functions : All	

Реализуйте пару интерфейсов/классов `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)
```

Вы должны реализовать класс `ProductsRepositoryJdbcImplTest`, содержащий методы, проверяющие функциональность репозитория, используя базу данных в памяти, упомянутую в предыдущем упражнении. На этом занятии следует заранее подготовить объекты модели, которые будут использоваться для сравнения во всех тестах.

Пример объявления тестовых данных приведен ниже:

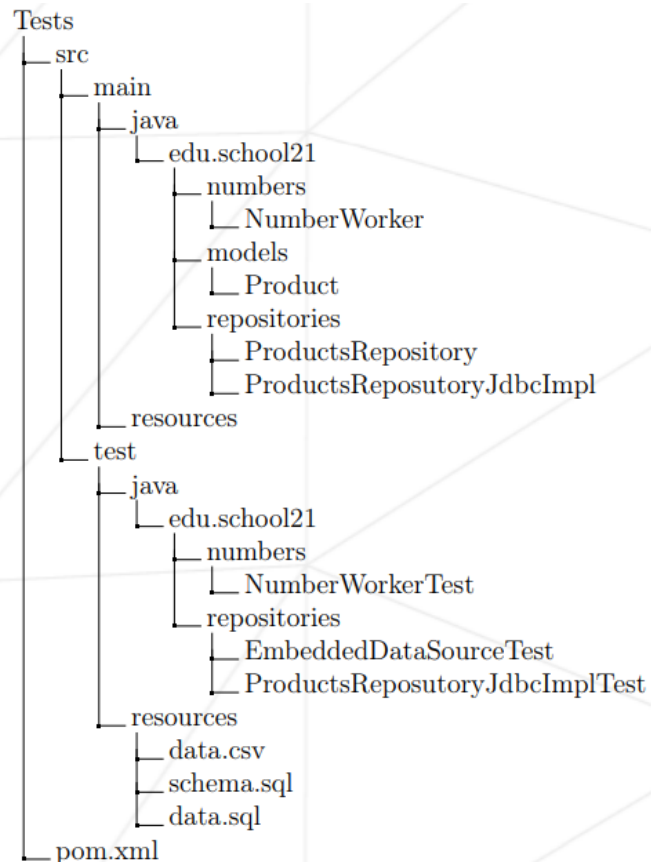
```
class ProductsRepositoryJdbcImplTest {
    final List<Product> EXPECTED_FIND_ALL_PRODUCTS = ...;
    final Product EXPECTED_FIND_BY_ID_PRODUCT = ...;
    final Product EXPECTED_UPDATED_PRODUCT = ...;
}
```

Заметки:

- Каждый тест должен быть изолирован от поведения других тестов. Таким образом, перед запуском каждого теста база данных должна находиться в исходном состоянии.
- Методы тестирования могут вызывать другие методы, которые не входят в текущий тест. Например, при тестировании метода `update()` может быть вызван метод `findById()` для проверки правильности обновления объекта в базе данных.




Структура проекта:



## Глава VII

### Упражнение 03. Проверка работоспособности

	Exercise 03
	Test for Service
	Turn-in directory : <i>ex03/</i>
	Files to turn in : Tests
	Allowed functions : All

Важное правило для модульных тестов: тестировать отдельный системный компонент, не вызывая функциональность его зависимостей. Такой подход позволяет разработчикам самостоятельно создавать и тестировать компоненты, а также откладывать реализацию конкретных частей приложения.

Теперь вам нужно реализовать слой бизнес-логики, представленный классом `UsersServiceImpl`.

Этот класс содержит логику аутентификации пользователя. Он также имеет зависимость от интерфейса `UsersRepository` (в этой задаче этот интерфейс реализовывать не нужно).

Интерфейс `UsersRepository` (описанный вами) должен содержать следующие методы:

```
User findByLogin(String login);  
void update(User user);
```

Предполагается, что метод `findByLogin` возвращает объект `Userobject`, найденный при входе в систему, или выдает исключение `EntityNotFoundException`, если пользователь с указанным именем пользователя не найден. Метод обновления выдает аналогичное исключение при обновлении пользователя, которого нет в базе данных.

Объект пользователя должен содержать следующие поля:

- Identifier
- Login
- Password
- Authentication success status (true - authenticated, false - not authenticated)

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

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

Этот способ:

1. Проверяет, прошел ли пользователь аутентификацию в системе с использованием этого логина. Если проверка подлинности была не выполнена, должно быть выброшено `AlreadyAuthenticatedException`.

2. Пользователь с этим логином извлекается из `UsersRepository`.

3. Если полученный пароль пользователя совпадает с заданным паролем, метод устанавливает для пользователя статус успешной аутентификации, обновляет его информацию в базе данных и возвращает `true`. Если пароли не совпадают, метод возвращает `false`.

Ваша цель:

1. Создайте интерфейс `UsersRepository`
2. Создайте класс `UsersServiceImpl` и метод аутентификации.
3. Создайте тест модуля для класса `UsersServiceImpl`.

Поскольку ваша цель — проверить правильность работы метода аутентификации независимо от компонента `UsersRepository`, вам следует использовать фиктивный объект и заглушки методов `findByLogin` и `update` (см. библиотеку `Mockito`).

Метод аутентификации должен быть проверен в трех случаях:

1. Правильный логин/пароль (проверьте вызов метода обновления с помощью инструкции по проверке библиотеки `Mockito`)
2. Неверный логин
3. Неверный пароль

Структура проекта:

