нетология

Тестирование программы. Mockito

Григорий
Вахмистров

Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. Вспоминаем прошлые занятия
2. Unit-тестирование
3. Что такое mock?
4. Библиотека Mockito
5. Итоги
6. Домашнее задание

Вспоминаем прошлые занятия

Вопрос 1

Какие уровни тестирования вы знаете?

Ответ

Какие уровни тестирования вы знаете?

1. Модульное
2. Интеграционное
3. Системное
4. Приемочное

Вопрос 2

Что тестируется в модульном тесте?

1. Классы
2. Методы классов
3. Взаимодействие модулей программы
4. Интеграцию программ между собой

Ответ

Что тестируется в модульном тесте?

1. **Классы**
2. **Методы классов**
3. Взаимодействие модулей программы
4. Интеграцию программ между собой

Вопрос 3

Какие аннотации для тестирования вы знаете?

Ответ

Какие аннотации для тестирования вы знаете?

1. @Test
2. @AfterAll/@AfterEach
3. @BeforeAll/@BeforeEach

Вопрос 4

Можно ли проверять с помощью JUnit появление/выброс исключений в методе?

1. Да
2. Нет

Ответ

Можно ли проверять с помощью JUnit появление/выброс исключений?

1. Да, используется вызов метода Assertions.assertThrows

2. Нет

Вопрос 5

Можно ли вызывать один и тот же тест с разными аргументами/параметрами?

1. Да
2. Нет

Ответ

Можно ли вызывать один и тот же тест с разными аргументами/параметрами?

1. Да

2. Нет

```
@ParameterizedTest
@ValueSource(strings = { "Hello", "World" })
public void testWithStringParameter(String argument) {
   Assertions.assertTrue(argument.contains("o"));
}
```

Unit-тестирование

На предыдущей лекции мы разобрали unit-тестирование (модульное). Сейчас мы рассмотрим наиболее частые проблемы, с которыми можем столкнуться во время создания тестов.

Давайте начнем с примера, не связанного с программированием.

Вопрос:

как бы вы, с точки зрения тестировщика, стали тестировать функционал приема денег и выдачи товара у автомата с Coca-Cola?

Тестирование автомата с Coca-Cola

Определенно, нужно было бы рассмотреть самые очевидные случаи в работе этого автомата - например, такие:

- выдача напитка за купюры
- выдача напитка за монеты
- возврат неправильных купюр
- микс правильных и неправильных монеток
- возврат монетки, если нет банок
- возврат монетки, если нет места для монеток

Тестирование автомата с Coca-Cola

Случаи, описанные на предыдущем слайде, предполагают тестирование всего автомата в сборе, а мы с вами хотим протестировать его модульно.

Вопрос:

Как вы думаете, как протестировать автомат с Coca-Cola **модульно**?

Разобрать?

Действительно, можно разобрать и протестировать каждый из модулей:

- монетоприемник
- механизм выдачи банки с напитком
- модули передачи информации*

* Например, информация о том, что автомат поврежден или что «Кока-Кола» в нем закончилась

Разобрать?

Каждый модуль может частично зависеть от другого модуля или каких-либо частей уже собранного автомата.

Например, все тот же монетоприемник зависит от сейфа, куда помещаются монеты, а механизм выдачи товара зависит от статуса монетоприемника - достаточное ли количество монет в него было опущено.

Тестируем зависимые модули вместе

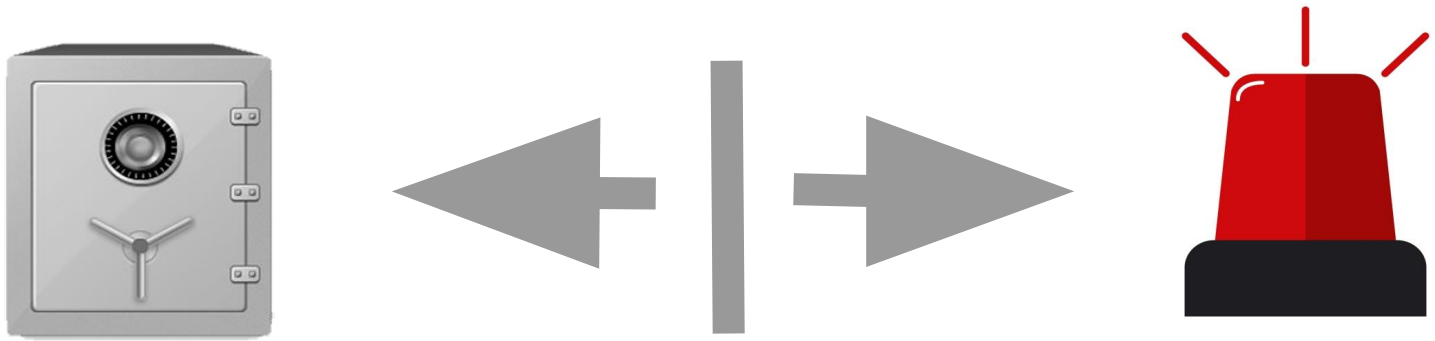
Таким образом, тестируя функционал модульно, мы сталкиваемся с проблемой тестирования, при которой один функционал зависит от другого.

Для решения такой проблемы придется **тестировать некоторые модули вместе** - например, сейф вместе с сигнализацией.

Тестируем зависимые модули вместе

А есть ли способы обойтись без настройки и подключения нашего модуля к другим - зависимым от него?

Забегая вперед, ответим, что варианты есть, а пока давайте рассмотрим еще один пример, уже из разработки.



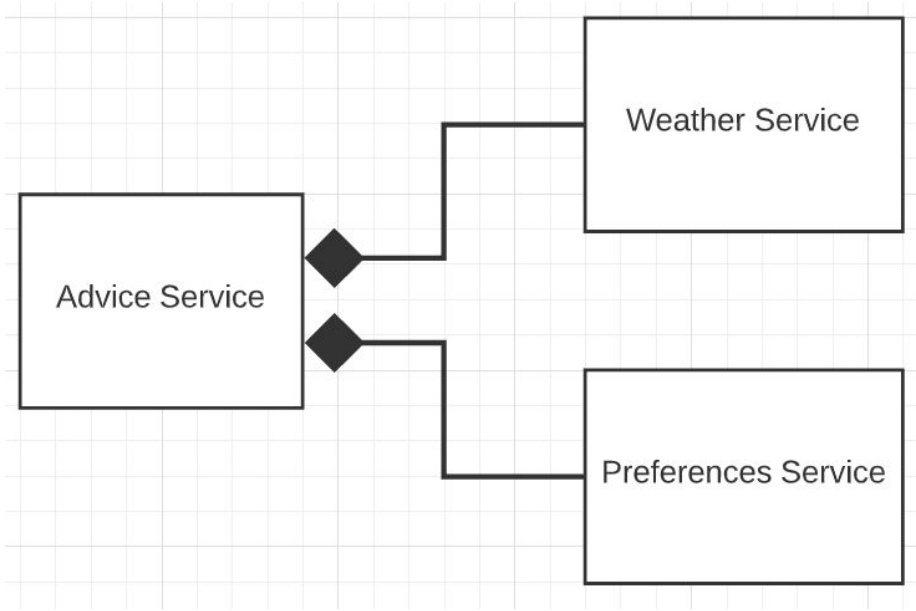
Пример: тестирование сервиса рекомендаций

Наша задача - протестировать сервис рекомендаций. Сервис советует вам, чем можно заняться в зависимости от погоды и ваших индивидуальных предпочтений.

На схеме изображен **Advice Service** - сервис рекомендаций, который зависит от 2-х других сервисов:

1) Сервис погоды (**Weather Service**) - возвращает нам актуальные данные о погоде в запрашиваемом городе

2) Сервис индивидуальных предпочтений пользователя (**Preferences Service**) - возвращает нам данные о том, чем любит заниматься пользователь, данные о его хобби и спортивных увлечениях.



Advice Service

```
import java.util.Set;
import java.util.stream.Collectors;

public class AdviceService {

  private final PreferencesService preferencesService;
  private final WeatherService weatherService;

  public AdviceService(PreferencesService preferencesService, WeatherService weatherService) {
    this.preferencesService = preferencesService;
    this.weatherService = weatherService;
  }

  public Set<Preference> getAdvice(String userId) {
    Weather weather = weatherService.currentWeather();
    Set<Preference> preferences = preferencesService.get(userId);
    if (Weather.RAINY == weather || Weather.STORMY == weather) {
      return preferences.stream()
        .filter(p -> p != Preference.FOOTBALL)
        .collect(Collectors.toSet());
    } else if (Weather.SUNNY == weather) {
      return preferences.stream()
        .filter(p -> p != Preference.READING)
        .collect(Collectors.toSet());
    }
    return preferences;
  }
}
```

Preferences Service

```
public interface PreferencesService {

   Set<Preference> get(String userId);
}

enum Preference {
   FOOTBALL("Сыграть в футбол"),
   WALKING("Выйти на прогулку"),
   WATCHING_FILMS("Посмотреть кино дома"),
   READING("Почитать книгу");

   private final String value;

   Preference(String value) {
      this.value = value;
   }
}
```

Weather Service

```
public interface WeatherService {

   Weather currentWeather();
}

enum Weather {
   RAINY("Дождливо"),
   STORMY("Сильный ветер"),
   SUNNY("Солнечно"),
   CLOUDY("Облачно");

   private String weather;

   Weather(String weather) {
       this.weather = weather;
   }
}
```

Потестируем Advice Service

Напишем тест, который проверяет, что сервис не посоветует нам пойти гулять в плохую погоду (в дождь или при сильном ветре).

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.Set;

class AdviceServiceTest {

   @Test
   void test_get_advice_in_bad_weather() {
      WeatherService weatherService = ?

      PreferencesService preferencesService = ?;

      AdviceService adviceService = new AdviceService(preferencesService, weatherService);
      Set<Preference> preferences = adviceService.getAdvice("user1");

      Set<Preference> expected = Set.of(Preference.READING, Preference.WATCHING_FILMS);
      Assertions.assertEquals(expected, preferences);
   }
}
```

Потестируем Advice Service

Чтобы тест заработал, знаки **“?”** нужно заменить на реализацию сервисов **WeatherService** и **PreferencesService**.

Мы можем использовать **вызовы реальных сервисов** - такой способ подходит, если:

- мы не боимся создать дополнительную нагрузку на эти сервисы для тестов
- мы уверены, что они будут гарантированно доступны там, где эти сервисы могут быть запущены

Это прямая аналогия с модульным тестированием автомата с «Кока-Колой», когда наш тестируемый модуль зависит от других модулей (сервисов).

Потестируем Advice Service

Недостатки вызова реальных сервисов:

- нужно, чтобы модули, от которых зависят тестируемые модули, были доступны и настроены
- ответ и обработка внешних модулей могут влиять на результат теста и замедлять его работу

Для решения такой проблемы в тестировании было введено понятие “**заглушка**” (или “**mock**” по-английски).

Что такое Mock?

Mock

Mock, или заглушка - это подражание поведению сервиса или объекта, функционал которого реализует mock.

Создание такой заглушки позволяет не использовать в тестах реальные модули, объекты или сервисы, которые часто могут зависеть от сети, баз данных, Интернета, а использовать копии этих модулей, возвращающих уже заранее подготовленный результат.

Mock

Создадим два mock, которые будем использовать в тестах:

- для **WeatherService** создадим класс-заглушку **WeatherServiceMock**
- для **PreferencesService** создадим класс-заглушку **PreferencesServiceMock**

WeatherServiceMock

```
public class WeatherServiceMock implements WeatherService {

    @Override
    public Weather currentWeather() {
        return Weather.STORMY;
    }
}
```

Такая класс-заглушка всегда будет возвращать только плохую погоду - добавим ей возможность изменять возвращаемое значение на наше:

```
public class WeatherServiceMock implements WeatherService {

    private Weather value;

    @Override
    public Weather currentWeather() {
        return value;
    }

    public void setValue(Weather value) {
        this.value = value;
    }
}
```

PreferencesServiceMock

```
import java.util.Set;

public class PreferencesServiceMock implements PreferencesService {

   private Set<Preference> value;

   @Override
   public Set<Preference> get(String userId) {
      return value;
   }

   public void setValue(Set<Preference> value) {
      this.value = value;
   }
}
```

На ваше усмотрение к таким классам можно добавить еще конструктор, который будет задавать значение по умолчанию для класса заглушки:

```
public PreferencesServiceMock() {
   this.value = Set.of(Preference.READING, Preference.FOOTBALL);
}
```

Подставим классы заглушки в тест

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.Set;

class AdviceServiceTest {

  @Test
  void test_get_advice_in_bad_weather() {
     WeatherServiceMock weatherService = new WeatherServiceMock();
     weatherService.setValue(Weather.STORMY);

     PreferencesServiceMock preferencesService = new PreferencesServiceMock();
     preferencesService.setValue(Set.of(Preference.FOOTBALL, Preference.WATCHING_FILMS,
Preference.READING));

     AdviceService adviceService = new AdviceService(preferencesService, weatherService);
     Set<Preference> preferences = adviceService.getAdvice("user1");

     Set<Preference> expected = Set.of(Preference.READING, Preference.WATCHING_FILMS);
     Assertions.assertEquals(expected, preferences);
  }
}
```

Какие проблемы возникают в таком подходе?

Создание **mock**-классов позволяет сократить зависимость тестируемого сервиса, но усложняет разработку тестов тем, что нам приходится на каждый зависимый модуль писать свои реализации.

В случае, если мы будем менять интерфейс любого из модулей, каждый раз придется актуализировать реализацию классов-заглушек.

Получается, при таком подходе нужно:

1. Создать класс-заглушку на каждый модуль, от которого зависит наш тест
2. Всегда актуализировать наши классы-заглушки в случае изменения базового класса

Звучит неплохо, но, может, есть способ писать меньше кода?

Действительно, можно писать меньше, используя уже разработанные библиотеки для тестирования. Одна из самых популярных библиотек называется **Mockito**.

Библиотека Mockito

Какие проблемы решает Mockito?

Библиотека Mockito позволяет упростить создание классов-заглушек. Нам не нужно будет каждый раз следить за изменениями в классе “родителей” любой из заглушек - достаточно будет использовать тот функционал, который необходим в тесте.

Перепишем наш тест с использованием Mockito

```
@Test
void test_get_advice_in_bad_weather() {
   WeatherService weatherService = Mockito.mock(WeatherService.class);
   Mockito.when(weatherService.currentWeather())
       .thenReturn(Weather.STORMY);

   PreferencesService preferencesService = Mockito.mock(PreferencesService.class);
   Mockito.when(preferencesService.get("user1"))
       .thenReturn(Set.of(Preference.FOOTBALL, Preference.WATCHING_FILMS, Preference.READING));

   AdviceService adviceService = new AdviceService(preferencesService, weatherService);
   Set<Preference> preferences = adviceService.getAdvice("user1");

   Set<Preference> expected = Set.of(Preference.READING, Preference.WATCHING_FILMS);
   Assertions.assertEquals(expected, preferences);
}
```

Mockito Spy

При создании тестов могут возникнуть ситуации, когда мы хотим использовать уже существующий объект в качестве mock или изменить только часть поведения модуля - сервиса. Для этого в **Mockito** вместо обычного метода **mock** нужно использовать метод **spy** (шпион), который создаст объект-заглушку, но будет передавать все свои вызовы реальному объекту, из которого был создан.

Создавать объект **Mockito.spy** из **interface** не имеет смысла, так как мы получим то же самое, что и при вызове Mockito.mock.

Реализация **Mockito.spy** происходит с помощью вызова метода **Mockito.mock**:

```
public static <T> T spy(Class<T> classToSpy) {
  return MOCKITO_CORE.mock(classToSpy, withSettings()
        .useConstructor()
        .defaultAnswer(CALLS_REAL_METHODS));
}
```

Пример Mockito Spy

Для проверки Mockito Spy создадим реализацию интерфейса:

```
public class WeatherServiceImpl implements WeatherService {

    @Override
    public Weather currentWeather() {
        return Weather.SUNNY;
    }
}
```

Вызовем в тесте и посмотрим, что вернет наш spy-объект:

```
@Test
void test_spy_weather_service() {
    WeatherService weatherService = Mockito.spy(WeatherServiceImpl.class);
    Assertions.assertEquals(Weather.SUNNY, weatherService.currentWeather());
}
```

У этого **spy**-объекта, как и у любого **mock,** можно переопределять значения, возвращаемые из методов, вызовом конструкции **when/thenReturn**

Mockito Verify

Помимо создания заглушек, библиотека Mockito позволяет узнать, сколько раз у нашей заглушки был вызван тот или иной метод во время теста. Для этого используется метод **Mockito.verify**.

Изменим наш первый тест и посмотрим, сколько раз был вызван метод **PreferencesService.get(String userId):**

```
@Test
void test_get_advice_in_bad_weather() {
  WeatherService weatherService = Mockito.mock(WeatherService.class);
  Mockito.when(weatherService.currentWeather()).thenReturn(Weather.STORMY);

  PreferencesService preferencesService = Mockito.mock(PreferencesService.class);
  Mockito.when(preferencesService.get(Mockito.any())).thenReturn(Set.of(Preference.FOOTBALL));

  AdviceService adviceService = new AdviceService(preferencesService, weatherService);
  adviceService.getAdvice("user1");

  Mockito.verify(preferencesService, Mockito.times(1)).get("user1");
  Mockito.verify(preferencesService, Mockito.times(0)).get("user2");
}
```

Mockito Verify

Verify принимает два аргумента:

1. Объект-заглушка
2. Объект, реализующий интерфейс Mockito - VerificationMode.

В нашем случае **Mockito.times** проверяет, сколько раз был вызван метод **get** с аргументом **“user1” и “user2”**. Так как вызовов со значением “user2” не было, мы проверяем **Mockito.times(0)**, такой вызов можно заменить на **Mockito.never()**.

Mockito Verify

Какие еще VerificationMode доступны:

- **Mockito.only()** - проверяет, что метод был вызван строго 1 раз
- **Mockito.atLeastOnce()** - проверяет, что метод был вызван хотя бы 1 раз
- **Mockito.atLeast(n)** - проверяет, что метод был вызван хотя бы “n” раз
- **Mockito.atMost(n)** - проверяет, что метод был вызван не более “n” раз
- **Mockito.timeout(int n)** - проверяет, что метод был вызван в течение заданного времени (“n” миллисекунд)
- **Mockito.after(int n)** - проверяет, что вызов метода был осуществлен после заданного интервала (“n” миллисекунд)

Mockito ArgumentCaptor

Библиотека позволяет получать значения, с которыми были вызваны методы mock-объекта. Для этого в библиотеке Mockito есть специальный класс **ArgumentCaptor**. Чтобы перехватить значение, переданное при вызове метода mock, нужно создать **ArgumentCaptor** с типом значения, которое мы будем перехватывать. Например, мы хотим получить аргумент, с которым был вызван метод **get** класса **PreferencesService**:

```
ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.forClass(String.class);
```

Я указал тип **String**, так как метод **get** принимает в качестве **userId** этот тип.

Следующим шагом нам нужно вызвать Mockito.verify и в качестве аргумента передать наш mock:

```
Mockito.verify(preferencesService).get(argumentCaptor.capture());
```

Для перехвата значения обязательно нужно вызвать метод **capture()**: он не изменит значение, но перехватит его и сохранит. Чтобы получить перехваченное значение у **argumentCaptor**, нужно вызвать метод **getValue()** или **getValues()**:

```
Assertions.assertEquals("user1", argumentCaptor.getValue());
```

Mockito ArgumentCaptor

Итоговый тест с проверкой переданного значения методу **get PreferencesService**

```
@Test
void test_get_advice_in_bad_weather() {
   WeatherService weatherService = Mockito.mock(WeatherService.class);
   Mockito.when(weatherService.currentWeather()).thenReturn(Weather.STORMY);

   PreferencesService preferencesService = Mockito.mock(PreferencesService.class);

Mockito.when(preferencesService.get(Mockito.any())).thenReturn(Set.of(Preference.FOOTBALL))
;
   ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.forClass(String.class);

   AdviceService adviceService = new AdviceService(preferencesService, weatherService);
   adviceService.getAdvice("user1");

   Mockito.verify(preferencesService).get(argumentCaptor.capture());
   Assertions.assertEquals("user1", argumentCaptor.getValue());
}
```

Mockito

Библиотека Mockito позволяет:

1. **Mock** - создавать mock-объекты (заглушки) и определять их поведение прямо в тесте
2. **Spy** - создавать объекты на основе классов реализаций и передавать вызовы реальным объектам
3. **Verify** - проверять, сколько раз тот или иной метод у объекта заглушки был вызван
4. **ArgumentCaptor** - проверять, с какими значениями были вызваны методы объекта заглушки

Ограничения Mockito

Что **не может** сделать библиотека Mockito:

- Не создает заглушки из final-классов
- Не создает заглушки к private-методам
- Не создает заглушки к static-методам
- Не создает заглушки к конструкторам
- Не создает заглушки к методам **equals** и **hashCode()**

Для более сложного создания объектов-заглушек можно рассмотреть библиотеку PowerMock, которая основана на Mockito, или рассмотреть аналоги EasyMock и JMock.

Выводы

Библиотека Mockito позволяет сократить время на создание объектов-заглушек, а также создавать и менять поведение этих объектов прямо в тестах.

Эта библиотека одна из самых популярных на текущий момент и рекомендуется к использованию.

Итоги

Рассмотрели и узнали:

- как можно использовать объекты-заглушки
- как писать изолированные модульные тесты
- как создавать объекты-заглушки с использованием библиотеки Mockito

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров