НЕТОЛОГИЯ

КОМПОЗИЦИЯ И ЗАВИСИМОСТЬ ОБЪЕКТОВ. MOCKITO ПРИ СОЗДАНИИ АВТОТЕСТОВ

ВАСИЛИЙ ДОРОХИН

ВАСИЛИЙ ДОРОХИН

QuadCode, QA Engineer

Василий Дорохин

План занятия

1. Задача
2. Зависимости
3. Mockito
4. Итоги

ЗАДАЧА

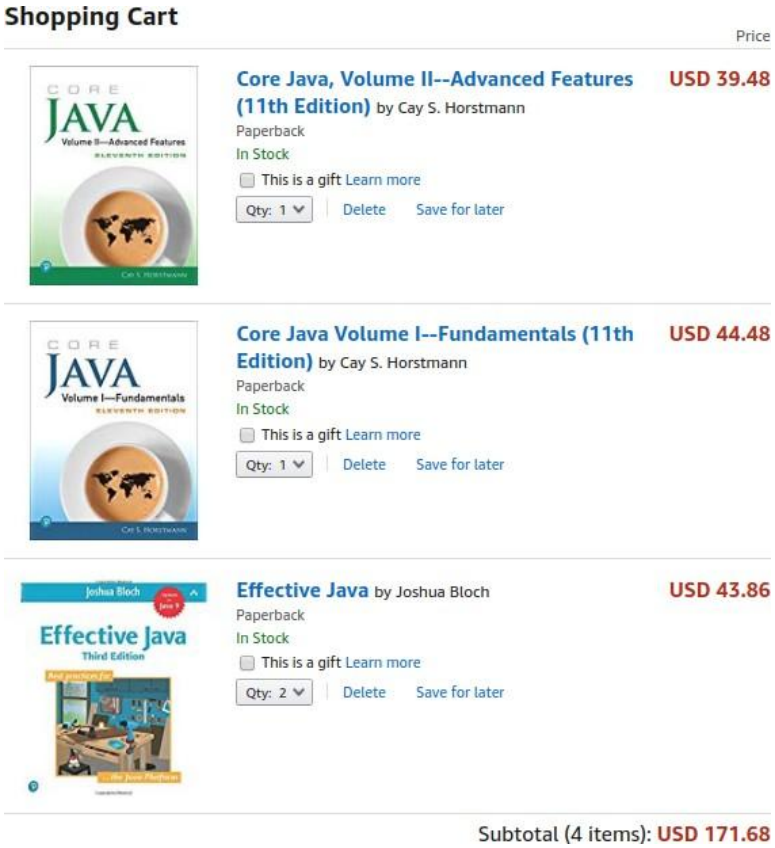
КОРЗИНА ПОКУПОК

Представим, что у нас есть сервис, на котором есть корзина покупок конкретного пользователя.

В корзину можно добавлять покупки, удалять, изменять количество определённой покупки.

При получении списка покупок самый последний добавленный элемент должен быть первым в списке (вот такие странные требования).

КОРЗИНА ПОКУПОК



Пример корзины сервиса Amazon.com. Именно там реализована логика, что последний добавленный товар показывается самым первым в корзине.

КОРЗИНА ПОКУПОК

Вопрос к аудитории: какие классы с какими полями и методами вы бы выделили для реализации подобной функциональности?

КЛАСС ИНФОРМАЦИИ О ПОКУПКЕ

```
public class PurchaseItem {
  private int id;
  private int productId;
  private String productName;
  private int productPrice;
  private int count;

  // + all args/no args constructor
  // + get/set на все поля
}
```

Вопрос к аудитории: как мы называли такие классы, предназначенные для хранения данных?

LOMBOK

```
@NoArgsConstructor
@AllArgsConstructor
@Data
public class PurchaseItem {
  private int id;
  private int productId;
  private String productName;
  private int productPrice;
  private int count;
}
```

ИНФОРМАЦИЯ О ПОКУПКЕ

Вопрос к аудитории: как вы считаете, зачем хранить цену продукта?

ИНФОРМАЦИЯ О ПОКУПКЕ

Всегда нужно сохранять данные товара на тот момент, в который пользователь положил его в корзину.

Иначе пользователь очень удивится, если он клал один продукт по одной цене, а потом вдруг в корзине оказался другой товар (если менеджер на сайте решил обновить название и/или изменить цену).

Примечание: кстати, это неплохой кейс для тестирования.

ДОБАВЛЕНИЕ ПОКУПКИ

Упрощённая реализация:

```
public class CartManager {
  private PurchaseItem[] items = new PurchaseItem[0];

  public void add(PurchaseItem item) {
    // создаём новый массив размером на единицу больше
    int length = items.length + 1;
    PurchaseItem[] tmp = new PurchaseItem[length];
    // itar + tab
    // копируем поэлементно
     for (int i = 0; i < items.length; i++) {
       tmp[i] = items[i];
     }
    // кладём последним наш элемент
    int lastIndex = tmp.length - 1;
    tmp[lastIndex] = item;
    items = tmp;
  }
```

for

```
for (инициализация; условие; пост.действия) {
    // выполняем блок кода, пока условие true
}
```

Типичный пример – перебор элементов массива:

```
// IDEA: itar + tab
for (int i = 0; i < items.length; i++) {
    System.out.println(items[i]);
}
```

i++ – операция инкремента, которая увеличивает значение i на единицу.

for

```
// IDEA: itar + tab
for (int i = 0; i < items.length; i++) {
    System.out.println(items[i]);
}
```

Что здесь происходит:

1. Инициализируется переменная i (счётчик цикла) в 0 (это переменная будет доступна только внутри цикла)

2. Проверяется i < items.length (true - идём далее, false - заканчиваем)

3. Обращаемся к элементу массива по индексу items[0] (т.к. i = 0) и печатаем

4. Выполняем пост.действия i++ (теперь i = 1)

5. Идём на пункт 2 (но уже с i=1) и так пока i < items.length == true

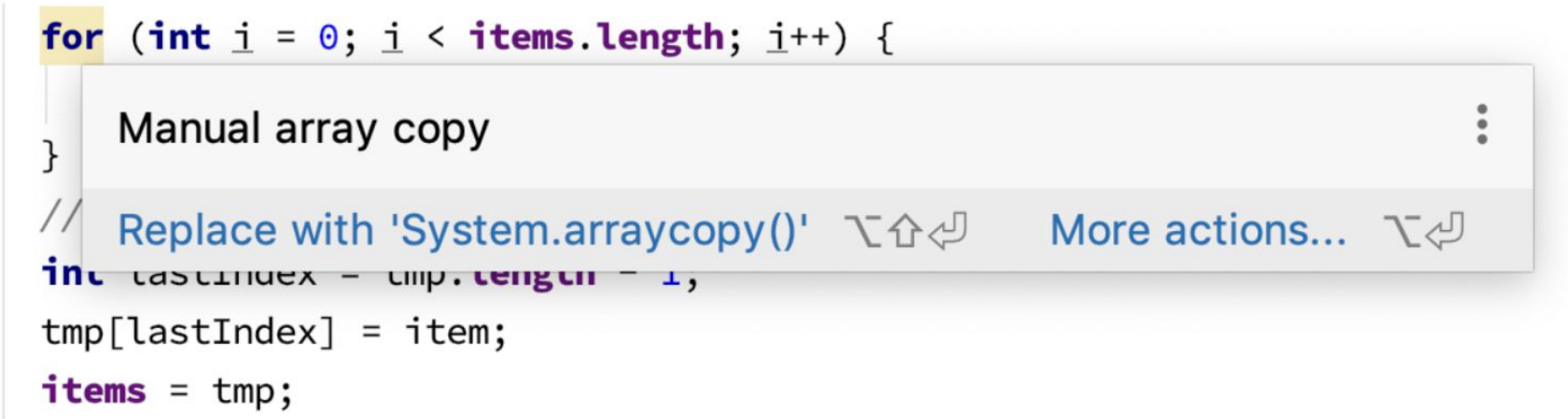
for

Продемонстрировать пошаговый проход цикла под отладчиком (точка остановки: System.out.println).

Особо обратить внимание на отслеживание изменения переменных при переходе на следующую итерацию.

ТИПОВОЙ КОД

IDEA распознаёт типовой код и выделяет его жёлтым цветом, предлагая более эффективные решения:



native

Если посмотреть на сигнатуру метода arraycopy, мы не увидим реализации:

```
public static native void arraycopy( @NotNull @Flow(...) Object src,  int  srcPos,
                                     @NotNull
                                     Object dest, int destPos,
                                     int length);
```

Ключевое слово native означает, что реализация этого метода написана не на Java.

Примечание*: про static мы будем говорить в следующей лекции.

СПИСОК ПОКУПОК

```
public class CartManager {
  private PurchaseItem[] items = new PurchaseItem[0];

  public void add(PurchaseItem item) {...}

  public PurchaseItem[] getAll() {
    PurchaseItem[] result = new PurchaseItem[items.length];
    // перебираем массив в прямом порядке
    // но кладём в результаты в обратном
    for (int i = 0; i < result.length; i++) {
      int index = items.length - i - 1;
      result[i] = items[index];
    }
    return result;
  }
```

УДАЛЕНИЕ ПО ID ПОКУПКИ

```
public class CartManager {
  private PurchaseItem[] items = new PurchaseItem[0];

  public void add(PurchaseItem item) {...}

  public PurchaseItem[] getAll() {...}

  // наивная реализация
  public void removeById(int id) {
    int length = items.length - 1;
    PurchaseItem[] tmp = new PurchaseItem[length];
    int index = 0;
    for (PurchaseItem item : items) {
      if (item.getId() != id) {
        tmp[index] = item;
        index++;
      }
    }
    // меняем наши элементы
    items = tmp;
  }
}
```

УПРОЩЕНИЯ

Нужно сделать следующее замечание: будем считать, что валидацией входных данных (в том числе проверкой достоверности цены) занимается другой сервис, который мы в данной лекции не рассматриваем.

Он осуществляет проверки:

1. Существования товара с productId в базе товаров;
2. Наличия достаточного количества на складах (как минимум, count штук);
3. Достоверности цены (цена в базе совпадает с переданной productPrice, в противном случае сообщает пользователю, что цена изменилась);
4. Созданием самого объекта PurchaseItem.

Кроме того, как вы видели, менеджер корзины не считает общую стоимость – когда мы будем рассматривать веб-приложения, мы поговорим, что это может быть переложено на плечи другого сервиса.

МЕНЕДЖЕР КОРЗИНЫ

Вопрос к аудитории: на какие большие группы вы бы смогли разделить тест-кейсы, в зависимости от текущего состояния корзины?

СОСТОЯНИЕ

В большинстве случаев для любой системы, содержащей набор элементов, можно выделить три ключевых состояния:

1. Элементов внутри системы не содержится;
2. Внутри системы есть ровно один элемент;
3. Внутри системы есть несколько элементов (больше одного).

СОСТОЯНИЕ

Q: Почему именно так?

A: Это один из возможных вариантов, ключевая идея которого, что именно эти состояния являются особыми.

1. Если в корзине нет элементов, то, массив результатов имеет нулевую длину.
2. Если в корзине всего один элемент, то нет смысла тестировать в каком порядке элементы выводятся на странице покупок, зато есть смысл протестировать добавление такого же товара и другого товара.
3. Если в корзине несколько элементов, то уже можно посмотреть на то, в каком порядке они будут отображаться при выводе (сортировка).

Примечание*: достаточно часто некоторые из этих сценариев можно схлопнуть в один.

Напишем несколько тестов на состояние, при котором у нас в корзине уже хранится несколько книг:

```
public class CartManagerTestNonEmpty {
  @Test
  public void shouldRemoveIfExists() {
    CartManager manager = new CartManager();
    int idToRemove = 1;
    PurchaseItem first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
    PurchaseItem second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
    PurchaseItem third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);
    manager.add(first);
    manager.add(second);
    manager.add(third);

    manager.removeById(idToRemove);

    PurchaseItem[] actual = manager.getAll();
    PurchaseItem[] expected = new PurchaseItem[]{third, second};

//    assertEquals(expected, actual);
    assertArrayEquals(expected, actual);
  }
```

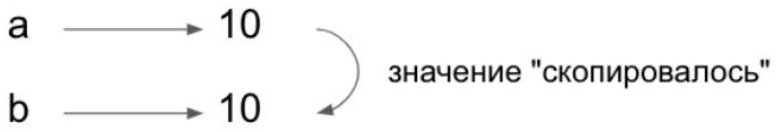
Q & A

Q: почему мы используем assertArrayEquals , а не assertEquals?

A: дело в том, что объекты - это ссылочные типы. Что это значит, давайте смотреть на картинке (см. следующий слайд).

ПРИМИТИВЫ

```
int a = 10;
int b = a;
```



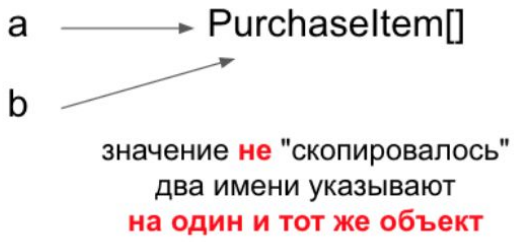

```
System.out.println(a == b);
```

сравниваем значения

ССЫЛОЧНЫЕ ТИПЫ

```
PurchaseItem[] a = new PurchaseItem[]
PurchaseItem[] b = a;

System.out.println(a == b);
```



В случае объектов (а массив – это объект), имена указывают (ссылаются на объект).

А когда мы создаём новое имя (b = a) имени, то объекты не копируются (т.е. не создаётся нового объекта), а новое имя ссылается на тот же объект.

ССЫЛОЧНЫЕ ТИПЫ

В случае массивов assertEquals будет сравнивать именно ссылки, а эти ссылки будут указывать на два разных объекта*:

```
// в методе getAll
PurchaseItem[] result = new PurchaseItem[items.length];
...
return result;

// в тесте:
PurchaseItem[] actual = manager.getAll();
PurchaseItem[] expected = new PurchaseItem[]{third, second};
```

Продемонстрировать, что при assertEquals тест падает.

assertArrayEquals

assertArrayEquals же сравнивает не сами массивы, а их элементы. При этом, если мы используем Lombok, то для нас генерируется специальный метод, который сравнивает уже не ссылки, а поля объектов. Как именно это происходит, мы узнаем с вами на следующей лекции.

ПОДГОТОВКА НАЧАЛЬНОГО СОСТОЯНИЯ

```
public class CartManagerTestNonEmpty {
  @Test
  public void shouldRemoveIfExists() {...}

  @Test
  public void shouldNotRemoveIfNotExists() {
    CartManager manager = new CartManager();
    int idToRemove = 4;
    PurchaseItem first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
    PurchaseItem second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
    PurchaseItem third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);
    manager.add(first);
    manager.add(second);
    manager.add(third);

    manager.removeById(idToRemove);

    PurchaseItem[] actual = manager.getAll();
    PurchaseItem[] expected = new PurchaseItem[]{third, second, first};

    assertArrayEquals(expected, actual);
  }
}
```

ArrayIndexOutOfBoundsException

Вопрос к аудитории: как вы думаете, почему второй тест падает?

ArrayIndexOutOfBoundsException

Мы уже сталкивались с таким термином как NPE (NullPointerException) – ситуацией, когда JVM аварийно заваершает работу приложения при попытке обратиться к null как к объекту.

В текущем же случае, мы пытаемся "положить" в массив размера 2 что-то по индексу 2 (а это не валидный индекс).

Пока мы будем считать это "ошибками" программы и чуть позже научимся с ними работать.

Вопрос к аудитории: давайте ещё раз посмотрим на наши тесты. Что вы можете сказать о них?

```
@Test
public void shouldRemoveIfExists() {
  CartManager manager = new CartManager();
  int idToRemove = 1;
  PurchaseItem first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
  PurchaseItem second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
  PurchaseItem third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);
  manager.add(first);
  manager.add(second);
  manager.add(third);

  manager.removeById(idToRemove);
```

```
@Test
public void shouldNotRemoveIfNotExists() {
  CartManager manager = new CartManager();
  int idToRemove = 4;
  PurchaseItem first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
  PurchaseItem second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
  PurchaseItem third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);
  manager.add(first);
  manager.add(second);
  manager.add(third);

  manager.removeById(idToRemove);
```

ДУБЛИРОВАНИЕ КОДА

Тесты проверяют разные кейсы: удаление существующего и несуществующего элемента, но настройка начального состояния идентична. Хотелось бы избежать дублирования кода.

Q: Почему бы нам просто не выстроить тесты в определённом порядке?

A: При написании unit-тестов исходят из того, что тесты не должны зависеть друг от друга и могут выполняться в произвольном порядке (в том числе параллельно). Хотя JUnit предоставляет соответствующие механизмы для упорядочивания тестов.

LIFECYCLE

JUnit нам предлагает целых 4-аннотации, которые позволяют что-то делать до тестов или после тестов:
— beforeAll и afterAll — методы, помеченные этими аннотациями, запускаются перед всеми тестами и после всех соответственно;
— beforeEach и afterEach — методы, помеченные этими аннотациями, запускаются перед каждым тестом и после каждого соответственно.

Напоминаем, что по умолчанию*, JUnit для выполнения каждого метода, отмеченного аннотацией @Test , создаёт новый объект, тем самым позволяя различным тестам не влиять друг на друга.

Примечание*: но это поведение тоже можно изменить.

LIFECYCLE

Таким образом:

```
public class CartManagerTestNonEmptyWithSetup {
  private CartManager manager;
  private PurchaseItem first;
  private PurchaseItem second;
  private PurchaseItem third;

  @BeforeEach
  public void setUp() {
    manager = new CartManager();
    first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
    second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
    third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);

    manager.add(first);
    manager.add(second);
    manager.add(third);
  }
```

LIFECYCLE

Совмещая с инициализацией поле, ещё короче:

```
public class CartManagerTestNonEmptyWithSetup {
  private CartManager manager = new CartManager();
  private PurchaseItem first = new PurchaseItem( id: 1,  productId: 1,  productName: "first",  productPrice: 1,  count: 1);
  private PurchaseItem second = new PurchaseItem( id: 2,  productId: 2,  productName: "second",  productPrice: 1,  count: 1);
  private PurchaseItem third = new PurchaseItem( id: 3,  productId: 3,  productName: "third",  productPrice: 1,  count: 1);

  @BeforeEach
  public void setUp() {
    manager.add(first);
    manager.add(second);
    manager.add(third);
  }
```

LIFECYCLE

```
public class CartManagerTestNonEmptyWithSetup {
  private CartManager manager = new CartManager();
  private PurchaseItem first = new PurchaseItem(1, 1, "first", 1, 1);
  private PurchaseItem second = new PurchaseItem(2, 2, "second", 1, 1);
  private PurchaseItem third = new PurchaseItem(3, 3, "third", 1, 1);

  @BeforeEach
  public void setUp() {...}

  @Test
  public void shouldRemoveIfExists() {
    int idToRemove = 1;
    manager.removeById(idToRemove);

    PurchaseItem[] actual = manager.getAll();
    PurchaseItem[] expected = new PurchaseItem[]{third, second};

//    assertEquals(expected, actual);
    assertArrayEquals(expected, actual);
  }
```

TEST HOOKS & HELPERS

Вспоминаем: иногда в Component Under Test (далее — CUT) вносят дополнительные инструменты, которые предназначены не для функционирования, а для того, чтобы систему можно было проверить или протестировать.

В нашем случае, мы можем попросить разработчика добавить в сервис корзины отдельный метод, который бы позволял «смотреть», что реально хранится в корзине (действительно ли после удаления элемент удаляется), а также метод, который за один раз позволяет добавить несколько покупок.

ЗАВИСИМОСТИ

ЗАВИСИМОСТИ

А что если наш сервис не сам хранит всю необходимую информацию, а берёт её из внешнего источника?

Например, из базы данных, внешнего сервиса или откуда-то ещё.

```
public class CartRepository {
  private PurchaseItem[] items = new PurchaseItem[0];

  public void save(PurchaseItem item) {...}

  public PurchaseItem[] findAll() { return items; }

  public void removeById(int id) {...}
}
```

ЗАВИСИМОСТИ

Грамотно построенный менеджер будет зависить от репозитория:

```
public class CartManager {
  private CartRepository repository;

  public CartManager(CartRepository repository) { this.repository = repository; }

  public void add(PurchaseItem item) { repository.save(item); }

  public PurchaseItem[] getAll() {...}

  public void removeById(int id) { repository.removeById(id); }
}
```

КЛЮЧЕВЫЕ МОМЕНТЫ

1. Менеджер получает репозиторий через конструктор, а не создаёт его внутри себя, т.е. мы можем при тестировании подставить туда нужную нам реализацию*.
2. Менеджер не должен (и ему не нужно) ничего знать о реализации.
3. Используется разделение ответственности между бизнес-логикой и хранением данных.

Мы уже говорили, что для удобного тестирования следует проектировать систему с учётом возможности тестирования.

Вопрос к аудитории: как называется степень, с которой система пригода к тестированию?

Примечание*: чуть позже мы будем говорить об интерфейсах, которые позволят организовать всё более гибко.

РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ

Разделение ответственности очень важно: т.к. тогда мы позволяем возможность "заменять" некоторые компоненты системы, не переписывая всю систему целиком.

Например, если мы поменяем внутреннюю реализацию CartRepository , то сам CardManager не изменится, т.к. он опирается на внешний "интерфейс"* и поведение репозитория.

Примечание*: под интерфейсом мы здесь понимаем набор публичных методов.

ИЗОЛИРОВАННОЕ ТЕСТИРОВАНИЕ

В реальной жизни: покупая в магазине лампочку, вы тестируете её прямо там, на стенде, чтобы удостовериться, что она сама по себе работает.

То же самое можно сказать про промышленные системы: части сначала тестируются изолированно (например, крыло самолёта) и только потом соединяются.

ЗАГЛУШКИ

Если мы тестируем только логику менеджера, нам нет смысла для этого реализовывать отдельную базу данных и т.д.

Мы же хотим только проверить, что если репозиторий отвечает определёнными данными, то менеджер будет их обрабатывать строго определённым образом:

```
manager = new CartManager(/* сюда нужно подставить заглушку CartRepository */);
```

Какие же инструменты есть для создания заглушки CartRepository?

MOCKITO

MOCKITO

Mockito — самая популярная библиотека для создания подобного рода заглушек:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.3.3</version>
    <scope>test</scope>
</dependency>
```

MOCKITO

```java
@ExtendWith(MockitoExtension.class) // расширение для JUnit
class CartManagerTestNonEmpty {
  @Mock // подставляет заглушку вместо реальной реализации
  private CartRepository repository;
  @InjectMock // подставляет заглушку в конструктор
  private CartManager manager;
  private PurchaseItem first = new PurchaseItem(1, 1, "first", 1);
  private PurchaseItem second = new PurchaseItem(2, 2, "second", 1);
  private PurchaseItem third = new PurchaseItem(3, 3, "third", 1);

  @BeforeEach
  void setUp() {
    // аналогично предыдущим тестам
  }
  ...
}
```

MOCKITO

```java
@ExtendWith(MockitoExtension.class) // расширение для JUnit
class CartManagerTestNonEmpty {
  ...
  @Test
  void shouldRemoveIfExists() {
    int idToRemove = 1;
    // настройка заглушки
    PurchaseItem[] returned = new PurchaseItem[]{second, third};
    doReturn(returned).when(repository).findAll();
    doNothing().when(repository).removeById(idToRemove);

    manager.removeById(idToRemove);
    PurchaseItem[] expected = new PurchaseItem[]{third, second};
    PurchaseItem[] actual = manager.getAll();
    assertArrayEquals(expected, actual);
    // удостоверяемся, что заглушка была вызвана с нужным значением
    // но это уже проверка "внутренней" реализации
    verify(cartRepository).removeById(idToRemove);
  }
  ...
}
```

MOCKITO

```
@ExtendWith(MockitoExtension.class) // расширение для JUnit
class CartManagerTestNonEmpty {
  ...
  @Test
  void shouldNotRemoveIfRemoveNotExists() {
    int idToRemove = 4;
    PurchaseItem[] returned = new PurchaseItem[]{first, second, third};
    doReturn(returned).when(repository).findAll();
    doNothing().when(repository).removeById(idToRemove);

    PurchaseItem[] expected = new PurchaseItem[]{third, second, first};
    PurchaseItem[] actual = manager.getAll();
    assertArrayEquals(expected, actual);
    // удостоверяемся, что заглушка была вызвана с нужным значением
    verify(repository).removeById(idToRemove);
  }
}
```

MOCKITO do*

Общая схема вызова методов do*(arg).when(Mock).method(args):

— doReturn(Object).when(Mock).method();

— doThrow(Throwable...).when(Mock).method()*;

— doThrow(Class).when(Mock).method()*;

— doNothing().when(Mock).method();

— doCallRealMethod().when(Mock).method().

Общая схема проверки вызова методов verify(Mock).method(args).

Примечание*: об исключениях мы будем говорить на одной из следующих лекций.

MOCKITO

Mockito умеет гораздо больше, чем мы описали, но наша задача в рамках этого курса — получить общее представление о том, что мы можем использовать заглушки (и будем) для целей тестирования.

MOCKING: ЗА И ПРОТИВ

Существуют разные мнения по поводу того, использовать Mock’и или нет.

В некоторых случаях излишнее увлечение ими может привести к тому, что вы начнёте тестировать реализацию, а не поведение, и работу заглушк, а не реальной системы.

В то же время, они очень помогают при unit-тестах, особенно при тестировании реакции объектов на различного рода исключительные ситуации, возникновения которых бывает сложно добиться.

UNIT-ТЕСТИРОВАНИЕ

Это и есть **юнит-тестирование**: мы тестируем объекты компоненты нашей системы изолированно, чтобы затем (когда мы их начнём соединять), быть уверенными, что сами по себе компоненты работают.

Это позволит быстрее проводить интеграцию.

ИТОГИ

ИТОГИ

Итак, мы рассмотрели достаточно много важных тем на сегодня:

— Композиция;

— Зависимости;

— Тестирование сервисов с зависимостями и Mockito.

ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше домашнее задание.

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

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

ВАСИЛИЙ ДОРОХИН

Василий Дорохин