COMPOSITION & DEPENDECY OF OBJECTS

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

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.

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.

Q: почему мы используем assertArrayEquals , а не assertEquals?

A: дело в том, что объекты - это ссылочные типы.

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 имяссылается на тот же объект.

ССЫЛОЧНЫЕ ТИПЫ

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

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.

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.

LIFECYCLE

JUnit нам предлагает целых 4-аннотации, которые позволяют что-то делать до тестов или после тестов:

1. beforeAll и afterAll — методы, помеченные этими аннотациями, запускаются перед всеми тестами и после всех соответственно;
2. 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, темсамым позволяя различным тестам не влиять друг на друга.

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.

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.

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:

Mockito — самая популярная библиотека для создания подобного
рода заглушек:

<dependency>
<groupid>org.mockito</groupid>
<artifactid>mockito-juint-jupiter</artifactid>
<version>3.3.3< /version>
<scope>test</scope>
</dependency>

MOCKITO do*

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

2. doReturn(Object).when(Mock).method();
3. doThrow(Throwable...).when(Mock).method()*;
4. doThrow(Class).when(Mock).method()*;
5. doNothing().when(Mock).method();
6. doCallRealMethod().when(Mock).method().

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

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.

Mocito code examples:

```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
  }
  ...
}
```

```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);
  }
  ...
}
```

```java
@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);
  }
}
```