top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

C++

</>

C++

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ

JAVA

Урок № 5

Использование JUnit и Mockito

Содержание

1. Модульное тестирование 4

2. JUnit 11

2.1. Назначение JUnit 11
2.2. Создание тестирующего класса в Eclipse 12
2.3. Структура фреймворка JUnit 16
2.4. Простой тест на положительный сценарий 18
2.5. Фикстуры 20
2.6. Тестирование исключительных ситуаций 24
2.7. Тестирование времени выполнения метода 27
2.8. Игнорирование тестов 28
2.9. Правила (Rules) 28
2.10. Наборы тестов 32
2.11. Параметризованные тесты 34
2.12. JUnit 5 38

3. Mockito 40
3.1. Назначение фреймворка Mockito 40
3.2. Настройка Eclipse для работы с Mockito 41
3.3. Описание тестируемой системы.....46
3.4. Создание стаба 48
3.4. Простой тест с использованием Mockito.....50
3.5. Основные возможности Mockito.....52
3.6. Контроль вызова методов в Mockito 55
3.7. Захват аргументов, передаваемых
в метод мока 58

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Модульное тестирование

Под тестированием кода понимается проверка соответствия между ожидаемым и реальным поведением программной системы. Тестирование может выполняться как самими программистами-разработчиками, так и специально обученными специалистами (тестировщиками).

В зависимости от анализируемых аспектов кода различают следующие виды тестирования:

1. **Функциональное тестирование** — проверка того, что код адекватно выполняет свою задачу и соответствует спецификации на него. Проверяется правильность работы всех методов кода: возвращаемые значения, выбрасываемые исключения и изменения в состоянии системы после выполнения каждого метода.
2. **Тестирование производительности** — оценка того, насколько быстро работает программа в обычных и в стрессовых условиях (например, при большом количестве пользователей интернет-магазин не должен замедлять свою работу или вообще становиться недоступным).
3. **Тестирование удобства использования** — обычно выполняется вручную специальным тестировщиком-юзабилистом. Такой тестировщик кликает по кнопкам, переходит по ссылкам и т.п., чтобы проверить работу интерфейса программы.
4. **Тестирование безопасности** — проверка того, что код не дает потенциальной возможности доступа

к несанкционированной информации, не позволяет испортить базу данных или препятствовать работе других пользователей, т.п.

Можно также провести классификацию по уровням тестирования:

1. **Модульное тестирование** (юнит-тестирование) — проверка работы отдельных модулей. Под модулем понимается обычно один класс или группа тесно взаимосвязанных классов. Такой модуль рассматривается изолировано. Если же он зависит от других частей программы (например, обращается к базе данных или к сетевому соединению), то на данном этапе зависимости закрываются специальными «заглушками». При этом считается, что окружение тестируемого модуля работает корректно. Этот вид тестирования обычно выполняется программистом-разработчиком класса. Обычно проверяется функциональность кода и иногда производительность.
2. **Интеграционное тестирование** — проверка совместной работы нескольких модулей (не обязательно пока всей системы в целом). Например, к оттестированному модулю добавляется реально работающая база данных. Цель этого этапа — проверить информационные связи между модулями. Этот вид тестирования может выполняться программистами или тестировщиками, в зависимости от политики руководства компании-разработчика.
3. **Системное тестирование** — это проверка работы системы в целом. Система должна быть помещена

в то окружение, где она будет эксплуатироваться, все компоненты должны быть реальными и уже прошедшими модульное тестирование. Обычно выполняется тестировщиками.

Когда говорят о тестировании, обычно выделяют два подхода:

1. **Тестирование «черного ящика»** — тестировщик не знает, как устроен код, а создает набор тестов только на основе спецификации к программе.
2. **Тестирование «белого ящика»** — тестировщик знает, как код устроен, и при разработке тестов может проверять, в том числе, некоторое внутреннее состояние системы, приватные методы, и т.п. Разумеется, в качестве тестировщика обычно выступает программист-разработчик кода.

В этом уроке мы будем говорить только о модульном тестировании (юнит-тестировании), которое должно сопровождать процесс разработки любого более-менее серьезного программного продукта. Понятно, что разработка тестов (а тесты — это, по сути, специальный код: классы и методы), требует определенных усилий и затрат времени. В чем же преимущества модульного тестирования, которые делают эти затраты оправданными?

- **Во-первых**, использование тестов поощряет программистов вносить изменения в код: добавлять новую функциональность или проводить рефакторинг. Если после внесения изменений предыдущие тесты выполняются успешно, то это служит доказательством того, что код по-прежнему работоспособен.

Это уменьшает панику и позволяет фиксировать работоспособные варианты продукта в системе контроля версий.

- **Во-вторых**, упрощается интеграция отдельных модулей. Если есть уверенность, что каждый модуль по-отдельности работоспособен, а при интеграции возникают ошибки — следовательно, дело в связях между ними, которые и нужно проверить.
- **В третьих**, тесты представляют собой особый вид документирования кода. Если программист-клиент не знает, как использовать данный класс, он может обратиться к тестам и увидеть там примеры использования методов этого класса.
- **В четвертых**, использование модульного тестирования повышает качество разрабатываемого кода. На большой и запутанный код трудно написать тест. Это заставляет программиста инкапсулировать отдельные фрагменты кода и отделять интерфейс от реализации. Методы тестируемого класса становятся проще и читабельнее.

С тестированием связан ряд приемов методологии экстремального программирования. Экстремальное **программирование** — это совокупность приемов организации работы программистов, которые позволяют сократить сроки разработки программного продукта, уменьшить стресс и в целом сделать процесс разработки более предсказуемым. Наиболее популярные методики экстремального программирования — это регрессионное тестирование и TDD (*разработка через тестирование*).

Регрессионное тестирование — это собирательное название для всех видов тестирования, направленных на обнаружение ошибок в уже протестированных участках кода. Если после внесения изменений в программу перестает работать то, что должно было продолжать работать, то возникает регрессионная ошибка (*regression bug*). Такие ошибки находятся, если после каждого изменения модуля прогоняется весь набор тестов, ранее созданных для этого модуля.

Рисунок 1. Алгоритм методики разработки через тестирование

Разработка через тестирование (TDD — *test-driven development*) — одна из практик экстремального программирования, которая предполагает разработку тестов до реализации кода. Т.е. сначала разрабатывается тест, который должен быть пройден, а потом — самый простой код, который позволит пройти этот тест. Алгоритм действий при реализации TDD показан на рисунке 1.

Таким образом, один цикл разработки методом TDD состоит в следующем:

1. Из репозитория извлекается модуль, на котором уже успешно выполняется некоторый набор тестов.
2. Добавляется новый тест, который не должен проходить (он может иллюстрировать какую-то новую функциональность или ошибку, о которой стало известно). Этот шаг необходим также и для проверки самого теста.
3. Изменяется программа так, чтобы все тесты выполнились успешно. Причем нужно использовать самое простое решение, которое не ломает предыдущие тесты.
4. Выполняется рефакторинг кода, после которого тесты тоже должны работать. Рефакторингом называется улучшение структуры кода без изменения его внешнего поведения. Например, переименовываются методы для лучшей читабельности программы, устраняется избыточный, дублирующий код, инкапсулируется поле, выделяется отдельный класс или интерфейс и т.п.
5. Весь комплект изменений вместе с тестами заносится в репозиторий (выполняется операция commit).

Таким образом, модуль всегда поддерживается в стабильном работоспособном состоянии.

Для каждого языка программирования существуют свои инструменты для создания и использования модульных тестов (юнит-тестов).

Инструменты модульного тестирования на Java:

1. Фреймворки (инфраструктуры) для написания и запуска тестов: *JUnit*, *TestNG*.
2. Библиотеки проверок: *FEST Assert*, *Hamcrest*, *XMLUnit*, *HTTPUnit*.
3. Библиотеки для создания тестовых дублеров: *Mockito*, *JMock*, *EasyMock*.

Библиотеки проверок позволяют расширить возможности проверок результатов тестов (которые имеются и в базовом фреймворке, например, в *JUnit*), а также сделать результаты логирования тестов более удобными.

Библиотеки для создания тестовых дублеров позволяют упростить написание «заглушек» для внешних по отношению к тестируемому модулей. Такие «заглушки» носят название моки (*mock-object*) и стабы (*stub-object*). **Stub** — более примитивный объект, просто заглушка. В лучшем случае может печатать трассировочное сообщение. **Mock** более интеллектуален и может реализовать какую-то примитивную логику имитации внешнего объекта.

В рамках данного урока мы рассмотрим два наиболее популярных инструмента: *JUnit* и *Mockito*.

2. JUnit

2.1. Назначение JUnit

JUnit — это один из наиболее популярных фреймворков для разработки юнит-тестов (модульного тестирования) на Java. Он был создан Кентом Беком и Эриком Гаммой (первая версия фреймворка для языка Smalltalk появилась в 1999 г). Аналогичные оболочки для модульного тестирования существуют и для других языков программирования. Они носят общее название — семейство XUnit.

Сегодня наиболее популярна версия JUnit 4. Предыдущая версия JUnit 3 предполагала жесткие правила задания имен тестирующих классов и методов, а также наследование от суперкласса TestCase. Вместо этого четвертая версия предполагает использовать аннотации.

Осенью 2017 года была анонсирована версия JUnit 5 (JUnit Jupiter), в которой изменились имена некоторых аннотаций и были добавлены ряд упрощений и дополнительных удобств.

Большинство современных IDE при создании тестового класса предлагает выбрать версию JUnit, с которой привык работать программист. В этом разделе все примеры ориентированы на JUnit 4, а в конце дается краткий обзор возможностей и отличий JUnit 5.

Как правило, модульные тесты разрабатываются с использованием IDE параллельно с разработкой кода. Затем они интегрируются в какую-либо систему управления версиями (ANT, Maven и т.п.) и запускаются регулярно в автоматическом режиме. Но иногда политика компани-

и-разработчика требует использования системы контроля версий с первого этапа работы над программным проектом. Для простоты в этом разделе далее используются встроенные возможности Eclipse для создания юнит-тестов.

2.2. Создание тестирующего класса в Eclipse

При использовании модульного тестирования важно грамотно сформировать структуру проекта: для тестов создается отдельная папка исходников (New/Source Folder) с именем tests. В ней дублируется структура пакетов папки src. Принято, чтобы каждый тестовый класс имел в конце своего имени слово Test. Например, если имеется класс Calculator, то для его тестирования создадим класс CalculatorTest.

Пример. Создание модульных тестов рассмотрим на примере класса Calculator, реализующего четыре арифметических действия:

```
package mycalc;
public class Calculator {
    public double add(double a, double b) {
        return a+b;
    }
    public double sub(double a, double b) {
        return a-b;
    }
    public double mult(double a, double b) {
        return a*b;
    }
    public double div(double a, double b) {
        return a/b;
    }
}
```

Для создания тестирующего класса из контекстного меню папки tests выберем New/JUnit Test Case. Появляется окно создания класса, показанное на рисунке 2.

В нем в верхней строчке переключателем задается версия JUnit. Имя тестирующего класса укажем CalculatorTest, а в нижней части окна указывается имя класса, для которого этот тест создается (Class under test). Можно нажать кнопку Browse рядом с этим полем и начать набирать имя класса. Eclipse предложит различные варианты имен на выбор.

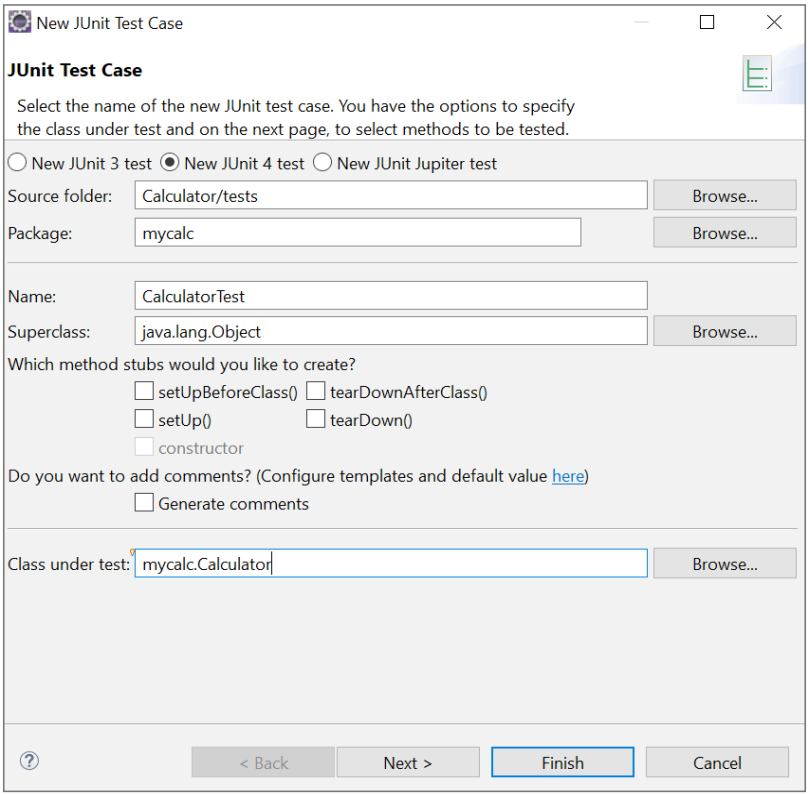

Рисунок 2. Окно создания тестового класса

Нажатие кнопки Next позволяет перейти к выбору методов этого класса, для которых будут созданы заготовки тестов. Отметим флажками все тестируемые методы (рисунок 3) и нажмем Finish для завершения.

Если создание тестового класса происходит впервые, Eclipse предложит добавить библиотеку JUnit 4 в проект (рисунок 4). А после согласия в структуре проекта появится строка *JUnit 4*.

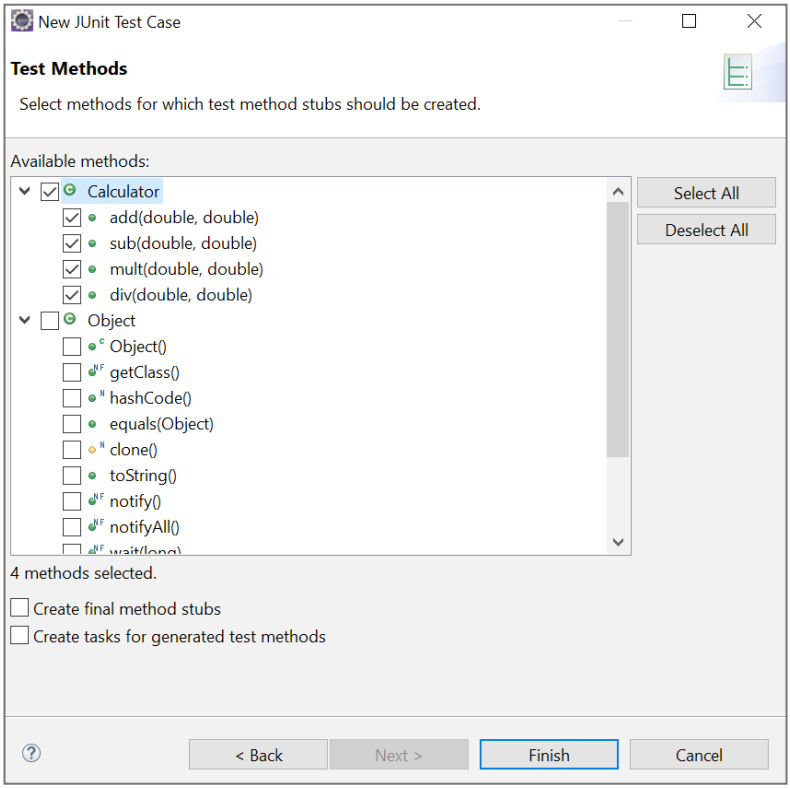

Рисунок 3. Выбор тестируемых методов

Созданный таким образом класс содержит заготовки тестирующих методов, как показано на рисунке 5. Все

имена тестирующих методов начинаются со слова test, хотя это и не обязательно. В JUnit 4 имена методов могут быть произвольными.

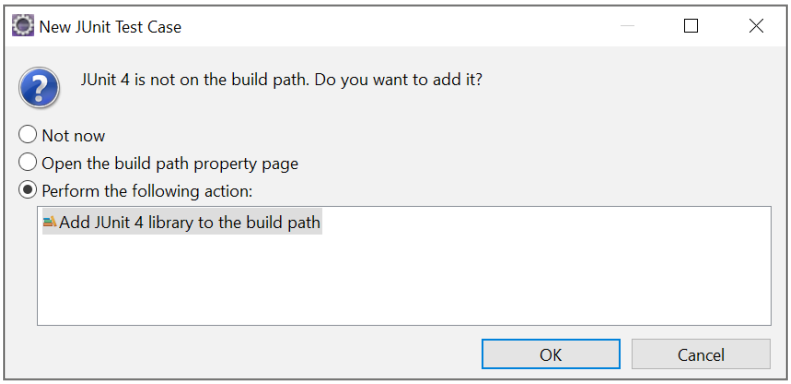

Рисунок 4. Добавление фреймворка JUnit в проект

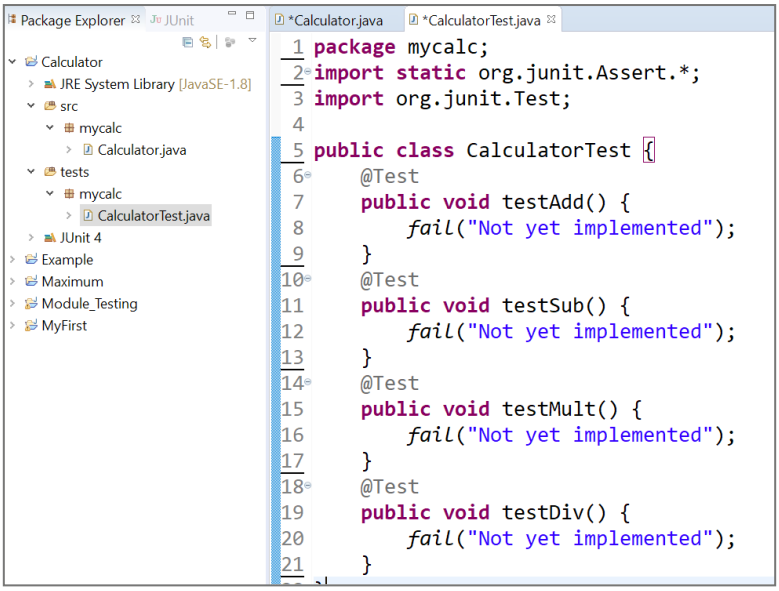

Рисунок 5. Заготовка тестирующего класса после создания

2.3. Структура фреймворка JUnit

Если посмотреть на импорты, которые были автоматически вставлены в код при создании тестирующего класса, то становятся очевидны две основные составляющие фреймворка JUnit:

1. Import org.junit. Test дает возможность использовать аннотацию Test. Аннотации — основной инструмент JUnit 4. Список наиболее часто используемых аннотаций приведен в таблице 5.1. Примеры их использования будут приведены далее по тексту.

Таблица 5.1 Аннотации JUnit 4

Аннотация	Описание
@Test	Определяет, что следующий за аннотацией метод является тестовым
@Before	Указывает, что следующий за аннотацией метод будет выполняться перед каждым тестом
@After	Указывает, что следующий за аннотацией метод будет выполняться после каждого теста
@BeforeClass	Указывает, что следующий за аннотацией метод будет выполнен перед всеми тестами в момент их запуска
@AfterClass	Указывает, что следующий за аннотацией метод будет выполнен после всех тестов
@Ignore	Указывает, что следующий метод будет пропущен (проигнорирован) в момент тестирования

2. Import static org.junit.Assert.* — подключение класса Assert, методы которого позволяют организовать различные проверки успешности прохождения тестов (таблица 5.2). Все методы класса Assert выбрасывают исключение AssertionError, если проверка не прошла. И тест при этом считается заваленным (failed).

Таблица 5.2 Некоторые методы класса Assert

Аннотация	Описание
fail ([String message])	Проваливает тест, выдавая текстовое сообщение (или без него)
assertTrue([String message,] boolean condition)	Проверяет, что логическое условие истинно
assertFalse([String message,] boolean condition)	Проверяет, что логическое условие ложно
assertEquals([String message], Object expected, Object actual)	Проверяет, что два значения объектов совпадают. Для примитивных типов проверяется обычное равенство, для объектов сравнивается содержимое методом equals(). Исключение — массивы (для них сравниваются ссылки, а не содержимое)
assertEquals([String message], double expected, double actual, double delta)	Проверяет, что два вещественных числа совпадают с заданной точностью
assertNull([String message], Object object)	Проверяет, что объектная ссылка является пустой (null)
assertNotNull([String message], Object object)	Проверяет, что объектная ссылка не является пустой
assertSame(([String message], Object expected, Object actual)	Проверяет, что обе ссылочные переменные относятся к одному объекту
assertNotSame(([String message], Object expected, Object actual)	Проверяет, что обе ссылочные переменные не относятся к одному объекту

Для сравнения массивов имеется метод AssertArrayEquals(), перегруженный для массивов разных типов.

В дополнение к классу Assert в JUnit имеется класс Assume. Методы этого класса при невыполнении условия

только выдают соответствующее сообщение, не генерируя никаких ошибок.

2.4. Простой тест на положительный сценарий

В первую очередь обычно создаются тесты, реализующие положительные сценарии работы тестируемых методов. Каждый тестовый метод предваряется аннотацией @Test. Имя тестового метода в JUnit 4 может быть любым, поэтому оставим без изменений имена testAdd, testSub и т.д., которые были автоматически созданы Eclipse.

Метод fail(), который был помещен в тела методов автоматически, делает тест проваленным (failed). Проваленный тест в окне JUnit обозначается рыжей линией, а успешный тест — зеленой. Провал теста говорит о том, что тестируемый метод работает не так, как мы предполагаем. Таким образом, все автоматически созданные тесты не проходят и при запуске дают рыжую линию. Это нужно для того, чтобы не пропустить заготовки и наполнить их настоящим тестовым содержанием. Обычно метод fail() комментируют или удаляют.

Тело теста должно соответствовать подходу AAA (arrange, act, assert). Это означает, что сначала выполняются некоторые подготовительные действия (arrange). Например, создается объект тестируемого класса. Затем запускается тестируемый метод (act). И, наконец, проверяется результат его работы (assert). Для такой проверки как раз и используются методы класса Assert. Ниже приведен пример теста, который проверяет выполнение сложения на примере 8+2=10 для метода add с целыми параметрами и целым результатом:

```
@Test
public void testAdd() {
   Calculator calc=new Calculator(); //arrange:
                                  // создание объекта
   int result=calc.add(8,2); // act: выполнение метода
   assertEquals(10, result); // assert: проверка
}
```

Однако в нашем тестируемом классе метод add объявлен с аргументами и результатом типа double. Вещественные числа представляются в памяти компьютера приблизительно. У чисел, которые мы считаем равными, может отличаться какой-нибудь шестой или десятый знак после запятой. Поэтому проверка на равенство или неравенство (== или !=) вещественных чисел может быть некорректна. Обычно проверку на равенство вещественных чисел заменяют проверкой того, что модуль их разности меньше очень маленького числа (отклонения). Например, в качестве отклонения можно взять 1e-9=10^{-9} (хотя это может быть и другое маленькое число 1e-6 или 1e-12 — все зависит от того, сколько знаков после десятичной точки мы должны учитывать при сравнении чисел). В JUnit для такого корректного сравнения вещественных чисел используется перегруженный вариант метода assertEquals() с дополнительным третьим параметром — отклонением:

```
@Test
public void testAdd() {
   Calculator calc=new Calculator(); // arrange:
                                  // создание объекта
```

```
    double result=calc.add(8,2); // act:
                                 // выполнение метода
    assertEquals(10, result,1e-9);//assert: проверка
}
```

Аналогично создадим тесты для методов sub(): 8-2=6, mult(): 8*2=16 и div(): 8/2=4.

Для запуска тестов достаточно нажать кнопку с зеленой стрелочкой на панели инструментов (или Run As/1 JUnit test из контекстного меню). Если все тесты успешно пройдены, то в окне JUnit будет показана зеленая полоска (рисунок 6). Если хотя бы один тест будет завален, то полоска будет рыжей.

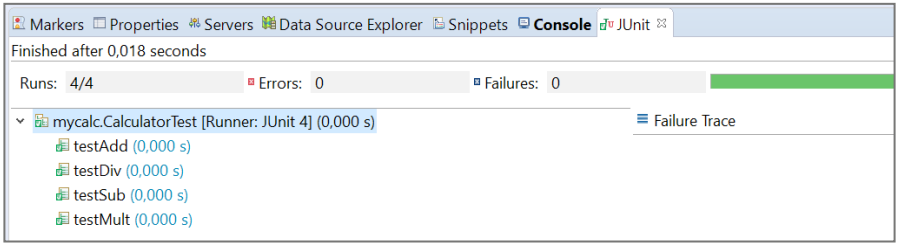

Рисунок 6. Окно результатов модульного тестирования

Обратите внимание на порядок тестов в окне *JUnit*. Он может быть произвольным, т.к. система выполняет каждый тест в отдельном потоке. Соответственно не гарантируется и определенный порядок завершения тестов.

2.5. Фикстуры

Фикстура — это состояние среды тестирования, которое нужно для выполнения теста. В нашем примере для каждого теста должен быть создан объект клас-

са Calculator. Чтобы упростить код, вынесем создание объекта в отдельный метод setUp(), который предварим аннотацией @Before:

```
public class CalculatorTest {
   private Calculator calc;
   @Before
   public void setUp() {
      calc=new Calculator(); // arrange: создание
                             // объекта
   }
   // тесты предполагают, что объект calc уже создан
}
```

Метод с аннотацией @Before будет выполняться перед каждым тестом. Таким образом, для каждого теста будет создан свой объект класса Calculator. Аналогично можно создать метод, который будет выполняться после каждого теста, задав перед ним аннотацию @After. Например, этот метод будет очищать ссылку calc:

```
package mycalc;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {
   private Calculator calc;
   @Before
   public void setUp() {
      calc=new Calculator(); // arrange: создание
                             // объекта
   }
```

```
    @After
    public void tearDown() {
        calc=null;  // удаление объекта после теста
    }

    @Test
    public void testAdd() {
        double result=calc.add(8,2); // act: выполнение
                                     // метода
        assertEquals(10, result,1e-9); // assert: проверка
    }

    @Test
    public void testSub() {
        double result=calc.sub(8,2);
        assertEquals(6, result,1e-9);
    }
      // Остальные тесты
}
```

Метод setUp() можно переписать с аннотацией @BeforeClass. Такой метод будет выполняться один раз перед всеми тестами. Т.е. объект calc будет создан один раз и использован во всех тестовых методах.

Нужно учесть, что метод с @BeforeClass должен быть статическим (соответственно и поле calc тоже)!

```
package mycalc;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalculatorTest {
    private static Calculator calc;
```

```
    @BeforeClass
    public static void setUp() {
        calc=new Calculator(); // arrange: создание
                               // объекта
    }

    @Test
    public void testAdd() {
        double result=calc.add(8,2); // act: выполнение
                                     // метода
        assertEquals(10, result,1e-9); // assert:
                                       // проверка
    }

    @Test
    public void testSub() {
        double result=calc.sub(8,2);
        assertEquals(6, result,1e-9);
    }
    // остальные тесты
}
```

Аналогично можно задать метод с аннотацией @AfterClass (тоже статический). И он будет выполняться после всех тестов.

Если бы при создании теста в окне Junit Test Case были поставлены флажки напротив setUp(), setUpBeforeClass(), tearDown(), tearDownAfterClass(), то мы бы получили соответствующие заготовки с аннотациями @Before, @BeforeClass и т.п. в файле тестового класса.

Можно задать несколько методов, помеченных аннотациями-фикстурами (@Before и т.д.) Однако порядок выполнения этих методов не гарантируется.

2.6. Тестирование исключительных ситуаций

Рассмотрим более внимательно метод деления двух вещественных чисел:

```
public double div(double a, double b) {
    return a/b;
}
```

Что произойдет, если второй параметр этого метода будет равен 0? Нет, исключение выброшено не будет. В Java исключение выбрасывается только при целочисленном делении на 0. А для вещественных чисел результатом будет константа NaN (*Not a Number*).

Если необходимо, чтобы метод все-таки выбрасывал исключение в этой ситуации, то его следует переписать следующим образом:

```
public double div(double a, double b)
                  throws DivByZeroException{
      if(b==0.0) throw new DivByZeroException ("Division
                                                by Zero");
      return a/b;
}
```

Также нужно создать собственный класс-исключение, унаследовав его от класса Exception:

```
package mycalc;

public class DivByZeroException extends Exception {
   public DivByZeroException() {
   }
```

```
    public DivByZeroException(String message) {
        super(message);
    }
}
```

Положительный сценарий работы метода div() проверяется тестом:

```
@Test
public void testDiv() throws DivByZeroException{
    double result=calc.div(8,2);
    assertEquals(4, result,1e-9);
}
```

Как же проверить, что в случае равенства нулю делителя исключение действительно выбрасывается (т.е. является правильной реакцией программы на исходные данные)? Нужно использовать аннотацию @Test с параметром expected:

- **@Test (expected=Exception.class)**

  Пример:

```
@Test(expected=DivByZeroException.class)
public void testDivByZero() throws DivByZeroException{
    double result=calc.div(8,0);
}
```

Такой тест будет пройден только в том случае, если исключение возникнет.

Еще один вариант проверки, связанный с исключением — это проверка того, что исключение не просто воз-

никло, но и выдало ожидаемое сообщение. В этом случае удобнее использовать аннотацию @Test без параметра:

```
@Test
public void testDivByZeroMessage() {
   try {
      double result=calc.div(8,0);
      fail("Division by Zero should have thrown
            a DivByZeroException");
      }
   catch(DivByZeroException e){
            assertEquals("Division by Zero",
            e.getMessage());
   }
   catch(Exception e) {
            fail("Should be DivByZeroException,
                  but have " + e.getClass());
   }
}
```

В приведенном выше примере в случае, если исключение не выбрасывается тестируемым методом, то тест заваливается методом fail() с сообщением о том, что должно быть исключение DivByZeroException.

Если же это исключение возникает, то оно отлавливается в первом блоке catch, и методом assertEquals() выполняется сравнение сообщения, инкапсулированного в объекте-исключении, с текстом «Division by Zero». Тест будет завален, если совпадения нет.

Если будет выброшено исключение другого класса, то это также приводит к неудаче теста с соответствующим выводом о несовпадении полученного исключения с ожидаемым.

2.7. Тестирование времени выполнения метода

В процессе модульного тестирования бывает очень важно оценить время выполнения метода. Например, если метод выполняет сетевое соединение, то слишком долгий таймаут означает, что это действие не выполнено или потеряны данные. Другой пример — игры или другие программы, работающие в реальном времени. Пользователь не должен слишком долго ждать отклика системы на его действия: это создает дискомфорт при использовании программы и мешает процессу игры. Для проверки длительности выполнения теста аннотация @Test имеет параметр timeout:

- **@Test (timeout=time)**

Здесь time — лимит времени выполнения теста (в миллисекундах). Если же тест выполняется дольше, то он считается проваленным.

Следующий тест будет провален (для его выполнения требуется больше времени, чем 10 миллисекунд):

```
@Test(timeout=10)
   public void testTimeOut() throws InterruptedException{
      for(long i=1;i<10;i++) {
      double result=calc.mult(i, i+1);
      double expected=i*(i+1);
      assertEquals(expected,result,1e-9);
      Thread.sleep(2);
   }
}
```

Если же параметр timeout установить равным 50, то тест будет успешно пройден.

Выполнение теста, превысившего лимит времени будет прервано, если это возможно (например, поток приостановлен методом sleep()). Если тест выполняет бесконечный цикл, то такой поток не остановится, и будет продолжать выполнение параллельно с выполнением других тестов.

2.8. Игнорирование тестов

При большом количестве тестов управление ими может вызвать серьезные затруднения: некоторые тесты устаревают, другие находятся в стадии разработки и еще не завершены. При этом пакет тестов прогоняется в системе управления версиями практически ежедневно.

Чтобы пропустить какой-либо тест во время выполнения очередного тестирования, используется аннотация @Ignore. В скобках после нее указывается комментарий с причиной игнорирования теста.

```
@Ignore("Test is not finished")
@Test
public void testSequence() {
   double result1=calc.add(3, 5);
   double result2=calc.mult(result1, 3);
}
```

2.9. Правила (Rules)

Правила позволяют гибко дополнять или переопределять поведение всех методов в тестирующем классе.

- **The TemporaryFolder Rule**

Это правило позволяет создавать файлы и папки, которые будут удалены по окончанию тестового метода

(независимо от того, пройден тест или нет). По умолчанию никакое исключение не выбрасывается, если эти временные папки не получается удалить.

Пример использования правила временных папок приведен далее по тексту. В тестовом методе testTempFile() во временный файл записывается таблица умножения на 3, полученная с помощью объекта класса Calculator. Затем эти числа прочитываются из файла, и происходит их сравнение с правильными значениями методом assertEquals().

```
@Rule
public final TemporaryFolder folder=new TemporaryFolder();
@Test
public void testTempFile() throws IOException{
   double result;
   File tempFile=folder.newFile();// задание файла во
                                  // временной папке
   FileWriter writer=new FileWriter(tempFile); //
                                  // поток записи
                                  // в файл
   for(int i=2;i<10;i++) {
       result=calc.mult(3, i); // вычисление
                               // произведения на 3
       String str=result+" ";
       writer.write(str); // запись в файл
   }
   writer.close(); // закрыть файл записи
   BufferedReader reader=
       new BufferedReader(new FileReader(tempFile));
   String str=reader.readLine(); // чтение всех данных
   reader.close();
   String[] numbers=str.split(" "); // разделение на
                                    // строки-числа
   for(int i=0;i<8;i++) {
```

```
    result=Double.parseDouble(numbers[i]); //
                            // преобразование в число
    assertEquals(3*(i+2),result,1e-9);// проверка
   }
}
```

- **The ExpectedException Rule**

Нам уже известен способ проверки того, что в методе должно возникнуть исключение — это параметр expected аннотации @Test. Создание правила ExpectedException дает еще большие возможности для проверки возникновения исключений: можно настроить проверку исключений сразу для всех тестов, и можно проверить сообщение, которое является параметром выбрасываемого исключения.

Сначала создается правило, задающее пустое проверяемое исключение. Поэтому те тесты, которые не должны выбрасывать исключения, успешно проходят.

```
@Rule
public final ExpectedException exception =
             ExpectedException.none(); // пустое
                                       // исключение
… // другие тесты без генерации исключений
```

Если же в тесте должно возникать исключение, то в его коде сначала настраивается класс ожидаемого исключения (методом expect()). Также можно настроить ожидаемый текст сообщения в этом исключении (метод expectMessage()).

Например, в приведенном ниже тесте проверяется возникновение исключения при делении на ноль. Также

проверяется, что это исключение содержит сообщение «Division by Zero»:

```
@Test
public void testDivByZero2() throws DivByZeroException
{
   exception.expect(DivByZeroException.class); //
                                // настройка ожидаемого
                                // класса исключения
   exception.expectMessage("Division by Zero"); //
                                // настройка ожидаемого
                                // сообщения в исключении
   double result=calc.div(8,0); // здесь появляется
                                   // исключение
}
```

Настройки исключения сбрасываются перед каждым тестом. Таким образом, для другого теста либо не ожидается исключение (пустое исключение согласно правилу), либо требуется настройка ожидаемого класса исключения.

- **The TimeOut Rule**

Еще один способ проверки времени выполнения теста. Позволяет задать общий лимит времени сразу для всех тестов.

Пример, ранее выполненный через параметр timeout аннотации @Test, при использовании правила выглядит следующим образом:

```
@Rule
public final Timeout globalTimeOut=Timeout.millis(200);
@Test
public void testTimeOut() throws InterruptedException{
   for(long i=1;i<10;i++) {
```

```
        double result=calc.mult(i, i+1);
        double expected=i*(i+1);
        assertEquals(expected,result,1e-9);
        Thread.sleep(2);
    }
}
```

С остальными правилами можно познакомиться, подробнее изучив Junit API (https://junit.org/junit4/javadoc/latest/).

2.10. Наборы тестов

Реальные приложения состоят из большого количества классов, для большинства из которых имеются соответствующие тестирующие классы. Как правило, все эти тесты должны регулярно прогоняться, чтобы контролировать работоспособность системы в процессе ее доработки и модификации.

Возникает необходимость объединять тестирующие классы и запускать их единым «набором». Для этой цели и предназначен класс Suite, позволяющий создать «запускалку» (runner) для нескольких тестирующих классов одновременно.

Для создания набора тестов нужно создать специальный класс, описание которого предваряется двумя аннотациями:

- @RunWith(Suite.class) — сообщает о том, что создается класс-раннер;
- @SuitClasses({TestClass1.class, TestClass2.class,…}) — перечисляет тестирующие классы, которые входят в данный набор.

Сам же класс оставляется пустым — он нужен только как контейнер для запускаемых классов.

Самый простой способ создать подобный набор в Eclipse — задать из контекстного меню папки test команду New/Junit Test Suite (если в меню команды New нет непосредственно этого варианта, то нужно выбрать пункт Other, а затем найти подходящий мастер настройки — wizard).

Появится окно создания набора, показанное на рисунке 7. В поле Test classes to include in suite можно отметить все тестирующие классы, которые должны быть включены в набор.

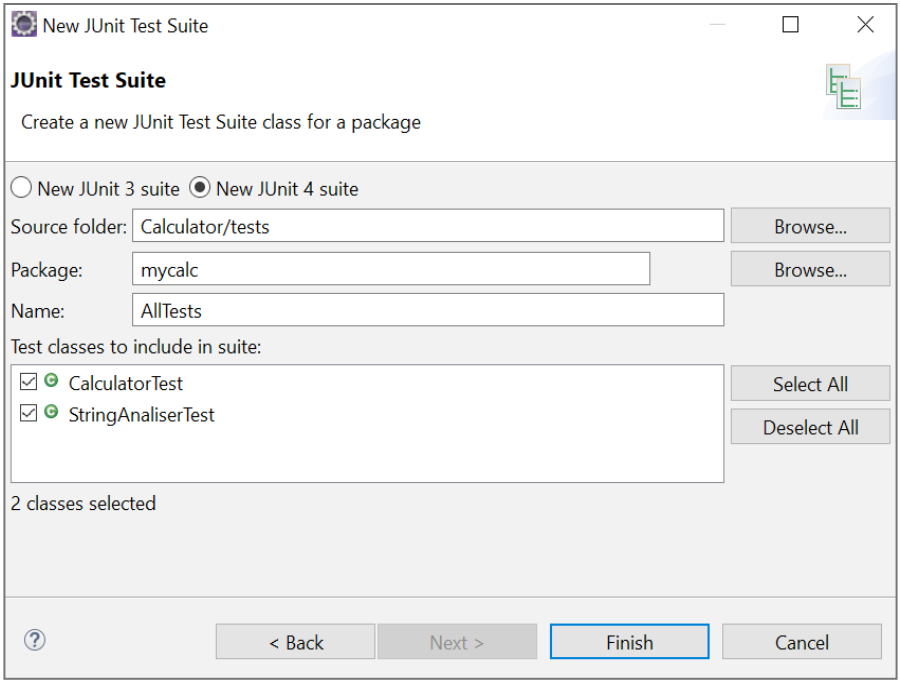

Рисунок 7. Окно создания набора тестов

В результате будет создан класс, код которого приведен ниже:

```
package mycalc;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ CalculatorTest.class,
StringAnaliserTest.class })
public class AllTests {
}
```

Запуск этого класса командой Run As приводит к запуску всех тестовых классов из набора.

2.11. Параметризованные тесты

При создании однотипных тестов возникает естественное желание автоматизировать этот процесс. Этой цели служат параметризованные тесты, в которых код теста остается неизменным, а меняются только данные, используемые для проверки.

Рассмотрим создание таких тестов на примере тестирования метода умножения mult().

1. Класс, который использует параметризованные тесты, должен предваряться аннотацией @RunWith с параметром Parameterized.class:

```
@RunWith(Parameterized.class)
public class CalculatorTest {
…//код тестирующего класса
}
```

2. В тестирующем классе обязательно должны быть параметры, представляющие набор данных для одного очередного теста (например, значения сомножителей и результат умножения). Также должен быть определен конструктор с параметрами, который задает значения этим полям:

```
@RunWith(Parameterized.class)
public class CalculatorTest {
   private double num1;
   private double num2;
   private double expectedResult;
   private static Calculator calc;
   public CalculatorTest(double num1, double num2,
                         double expectedResult) {
      this.num1 = num1;
      this.num2 = num2;
      this.expectedResult=expectedResult;
   }
…
}
```

3. В тестирующем классе также должен быть статический метод, который возвращает список наборов данных (значений полей) для всех тестов. Этот список должен предваряться аннотацией @Parameters. В скобках аннотации указывается параметр — имя, позволяющее идентифицировать тесты.

```
@Parameters(name="{index}:mult({0},{1})= {2}")
public static Collection<Object[]> dataForTests(){
   return Arrays.asList(new Object[][] {
         {2,3,6},{2,7,14},{0,3,0}});
}
```

Имя теста может и отсутствовать. В примере оно включает следующие элементы: {index} — номер теста (начинается с нуля); {0}, {1} и {2} — значения полей, содержащих исходные данные для теста (по порядку их объявления в классе). Остальное в имени — поясняющий текст, который так и выводится. В результате после прогона параметризованного теста в окне JUnit будет выведено сообщение, показанное на рисунке 8.

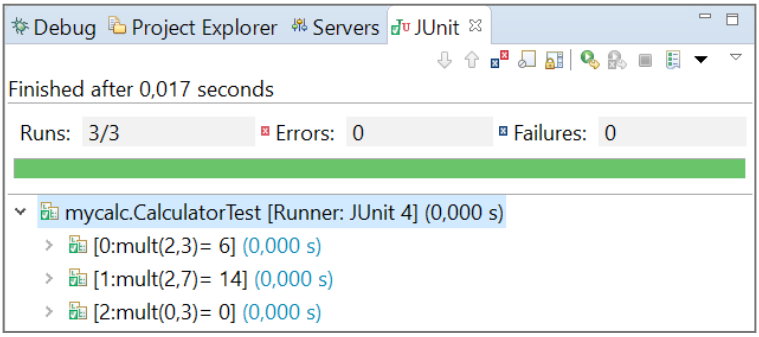

Рисунок 8. Результаты выполнения параметризованного теста

Что касается тела метода, то статический метод asList() класса Arrays преобразует массив в коллекцию и дает удобный способ задать элементы коллекции в коде программы.

1. И, наконец, собственно тест на умножение просто использует поля класса для задания исходных данных и результата умножения:

```
@Test
public void testMult() {
    double result=calc.mult(num1,num2);
    assertEquals(expectedResult, result,1e-9);
}
```

В целом код класса с параметризованными тестами представлен ниже:

```
package mycalc;
import static org.junit.Assert.*;

@RunWith(Parameterized.class)
public class CalculatorTest {
   private double num1;
   private double num2;
   private double expectedResult;
   private static Calculator calc;

   public CalculatorTest(double num1, double num2,
                         double expectedResult) {
      this.num1 = num1;
      this.num2 = num2;
      this.expectedResult = expectedResult;
   }

   @Parameters(name = "{index}:mult({0},{1})= {2}")
   public static Collection<Object[]> dataForTests(){
      return Arrays.asList(new Object[][] {
             {2,3,6},{2,7,14},{0,3,0}});
   }

   @BeforeClass
   public static void setUp() {
      calc=new Calculator();
   }

   @Test
   public void testMult() {
      double result=calc.mult(num1,num2);
      assertEquals(expectedResult, result,1e-9);
   }
```

Необходимо отметить, что параметры для теста передаются один раз на итерацию, а не перед вызовом каждого теста. Другими словами, если тестирующий класс содержит несколько тестов, то очередной набор параметров устанавливается перед запуском всех тестов в классе, а не перед каждым тестом индивидуально. По этой причине тесты не должны изменять значения полей, которые используются как параметры.

2.12. JUnit 5

Последняя версия фреймворка JUnit 5 появилась в конце 2017 г. Она ориентирована на Java 8. Однако можно проверить и код, который был скомпилирован с предыдущими версиями языка.

В JUnit 5 изменились названия некоторых аннотаций. В таблице 5.3 показано соответствие новых и старых названий.

Таблица 3. Соответствие названий аннотаций в JUnit 4 и JUnit 5

JUnit 4	JUnit 5
@BeforeClass	@BeforeAll
@Before	@BeforeEach
@AfterClass	@AfterAll
@After	@AfterEach
@Ignore	@Disabled

Появились новые аннотации, которые улучшают читаемость кода и позволяют упростить разработку тестов. Некоторые из них приведены в таблице 5.4.

Таблица 4. Новые аннотации в Junit 5

Аннотация	Назначение
@DisplayName	Улучшение читаемости при выводе результатов теста
@Nested	Возможность использовать вложенные классы при разработке тестов
@RepeatedTest	Помеченный этой аннотацией тест можно запустить несколько раз. При этом каждый вызов будет независимым тестом, т.е. для него будут работать аннотации @BeforeAll, @BefoireEach, @AfterAll, @AfterEach
@ParameterisedTest	Упрощение создания параметризованных тестов
@ValueSource @ArgumentSource @CsvSource @EnumSource	Применяются для задания источника данных параметризованного теста (после аннотации @ParameterisedTest)

Также были добавлены ряд новых методов: assertAll() — для логической группировки тестов, assertIterableEquals() — для работы с Iterable-объектами, assertLinesMatch() — метод для сравнения набора строк с поддержкой регулярных выражений и др.

Подробнее о возможностях Junit 5 можно узнать в руководстве пользователя: https://junit.org/junit5/docs/current/user-guide/.

3. Mockito

3.1. Назначение фреймворка Mockito

Модульное тестирование предполагает изоляцию исследуемого класса от его окружения. Но на деле реализовать эту изоляцию непросто. Например, для работы тестируемого класса нужны данные, которые ему предоставляет другой класс (берет их из базы данных или от внешнего Web-сервиса).

Во время тестирования нужно как-то «подменить» работу этого внешнего класса некоторой простой «заглушкой». Это даст уверенность в том, что окружение работает корректно и предсказуемо, и что обнаруживаемые ошибки сосредоточены именно в тестируемом модуле. Кроме того, во время тестирования одного модуля взаимодействующие с ним классы могут быть еще не созданы или не до конца отлажены.

Принято выделять два основных вида «заглушек» (или тестовых дублеров): стабы (*stub*) и моки (*mock*). Для создания стабов не обязательно использовать дополнительные фреймворки, их можно реализовать средствами Java Core. При этом работа программиста будет достаточна сложна, а полученный результат имеет ограниченные возможности применения.

Для создания моков на Java можно использовать фреймворк Mockito. С его помощью легко и динамично создаются объекты, имитирующие окружение тестируемого модуля. Функциональность таких объектов

значительно выше, чем у стабов, а работы программисту — меньше.

В данном разделе будет описана простая тестируемая система. На ее примере мы сначала создадим стаб, а затем — мок с использованием Mockito. Таким образом, будут продемонстрированы возможности и преимущества этого фреймворка.

3.2. Настройка Eclipse для работы с Mockito

При использовании различных сторонних библиотек в Java-проекте всегда возникает ряд проблем. Где скачать эту библиотеку? Какую версию выбрать? Как подключить библиотеку к проекту, т.е. сделать так, чтобы компилятор находил нужные классы из этой библиотеки?

Наиболее комфортный способ решения этих проблем — это использовать **Maven** — один из самых популярных инструментов для сборки проектов на Java. Он отвечает за подключение фреймворков и всех необходимых классов проекта, сборку и создание исполняемых jar-файлов, а также за генерацию документации. Кроме того, Maven заставляет программиста иметь стандартную грамотную структуру каталогов в проекте. Альтернативные инструменты — Ant и Gradle, но они менее популярны.

И хотя при работе с JUnit мы до сих пор обходились без этого инструмента (и для Mockito это тоже возможно), теперь все же создадим Maven-проект и воспользуемся его преимуществами.

Для того, чтобы использовать Maven, при создании проекта укажем File/New/Maven Project. Если этого пункта нет в меню команды New, то следует сначала

выбрать Other, а затем найти нужный мастер создания проекта. Альтернативный вариант: File/New/Project/Maven/Maven Project.

В появившемся окне (рисунок 9) нужно установить флажки

- Create a simple project;
- Use default workspace location (и затем указать папку, в которой будет размещен создаваемый проект)

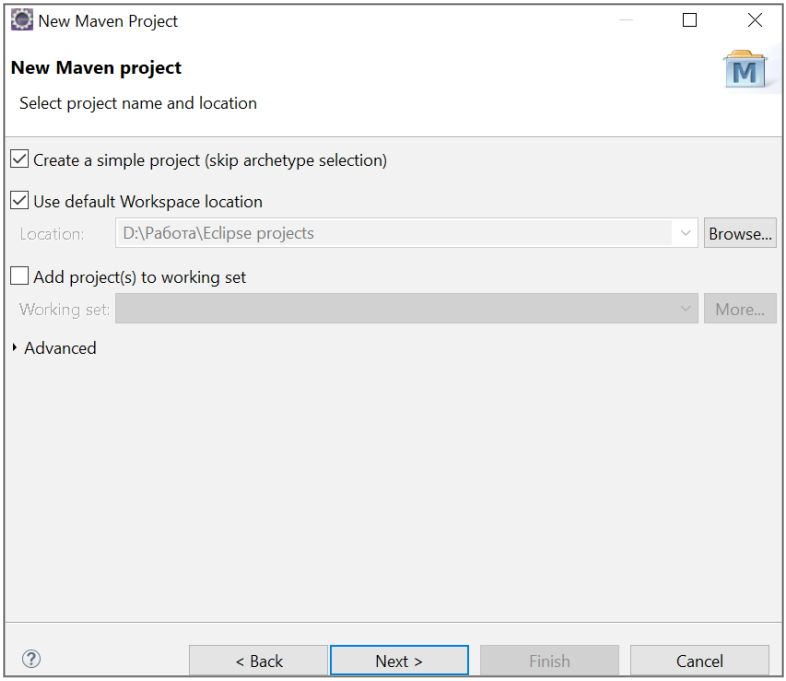

Рисунок 9. Окно создания Maven проекта

Щелчок по кнопке Next открывает следующее окно (рисунок 10), в котором будут заданы основные параметры проекта. Например, в поле Artifact Id указывается имя проекта (mockito-example).

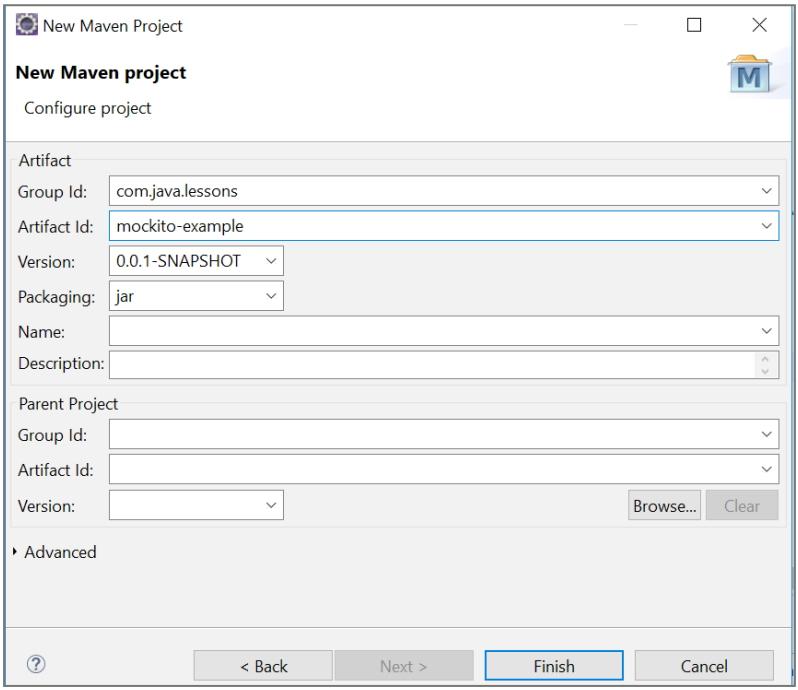

Рисунок 10. Окно конфигурирования проекта

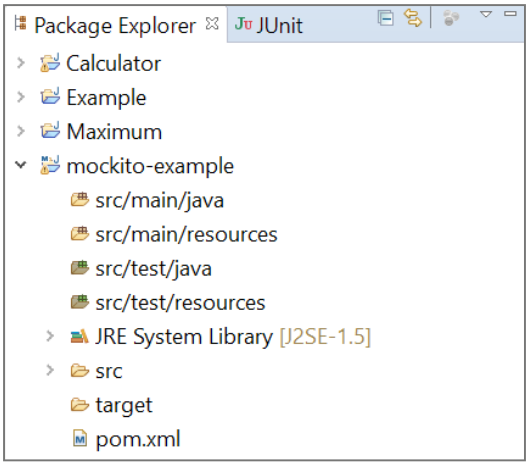

Рисунок 11. Структура Maven проекта

Структура созданного проекта показана на рисунке 11. Исходный код классов следует помещать в папку src/main/java, а тесты — в папку src/test/java.

В типичном Maven проекте имеется файл *pom.xml*, в котором дана информация о проекте и о деталях его конфигурации. Содержимое этого файла, автоматически созданное Eclipse, показано на рисунке 12.

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://ww
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.java.lessons</groupId>
4   <artifactId>mockito-example</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6 </project>
```

Рисунок 12. Автоматически созданный файл pom.xml

В данном модуле мы не будем подробно разбирать теги XML. Чтобы использовать JUnit и Mockito, просто добавьте перед закрывающим тегом </project> следующий код:

```
<dependencies>
     <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
     </dependency>
     <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.10.19</version>
        <scope>test</scope>
     </dependency>
  </dependencies>
```

Заданные таким образом зависимости будут подгружены к проекту. Появится папка Maven Dependencies, которая содержит все необходимые библиотеки (рисунок 13).

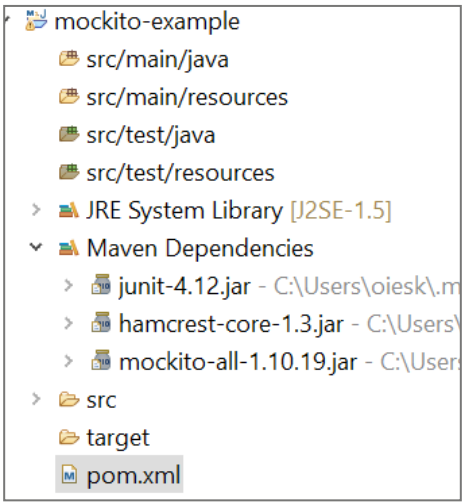

Рисунок 13. Структура проекта, настроенного на использование Junit и Mockito

Еще один момент в настройке Eclipse состоит в том, чтобы упростить автоматическую генерацию статических импортов в проекте (это то, предлагается в контекстном меню в случае, если класс явно не указан).

Зайдите в настройки Window/Preferences/Java/Editor/Content Assist/Favorites. Должны быть прописаны следующие классы:

- org.junit.Assert
- org.mockito.BDDMockito
- org.mockito.Mockito
- org.hamcrest.Matchers
- org.hamcrest.CoreMatchers

Если какого-либо описания не хватает, добавьте его с помощью кнопки New Type.

3.3. Описание тестируемой системы

Применение Mockito рассмотрим на простом примере. Тестируемая система обычно обозначается термином SUT — *System Under Test*.

Пусть в этой системе происходит обслуживание некоторых заданий пользователей (каких — в данном случае неважно). Эти задания поступают от некоторого класса, реализующего интерфейс TodoService с единственным методом retrieveTodos(), возвращающим список задач пользователя:

```
public interface TodoService {
    // метод возвращает список задач пользователя
    public List<String> retrieveTodos(String user);
}
```

Использование интерфейса — это эффективный прием, позволяющий легко подменять заглушкой настоящий класс.

Класс TodoBusinessImpl, который необходимо тестировать, реализует некоторую бизнес-логику по обслуживанию заданий пользователя. Для простоты сделаем в этом классе один метод, который должен из всего списка задач отобрать те, в названии которых содержится подстрока «Spring».

```
public class TodoBusinessImpl {
// класс реализует какую-то бизнес-логику на основе
// задач пользователя
```

```
    private TodoService todoService; // приватное
                                     // поле — список задач

    public TodoBusinessImpl(TodoService todoService) {
        this.todoService = todoService;
}

// метод, который будем тестировать: отбирает среди
// задач пользователя те, которые относятся к Spring
    public List<String>
            retrieveTodosRelatedToSpring(String user){
        List<String> filteredTodos=new ArrayList<String>();
        // вызов метода, реализации которого еще нет
        List<String> todos=todoService.retrieveTodos(user);
        for(String todo:todos) {
            if(todo.contains("Spring")) {
                filteredTodos.add(todo);
            }
        }
        return filteredTodos;
    }
}
```

Проблема тестирования класса TodoBusinessImpl заключается в том, что вызываемый в коде метод retrieveTodos() еще не реализован и его работа во время тестирования должна быть имитирована стабом или моком.

Согласно логике приложения создадим два пакета в папке src/main/java: businessLogic — для тестируемого класса и data — для интерфейса TodoService (рисунок 14). Аналогичные пакеты должны быть в папке src/test/java. Разрабатываемые тесты в дальнейшем мы будем помещать в пакет businessLogic папки test, а стаб расположим в пакете data этой же папки.

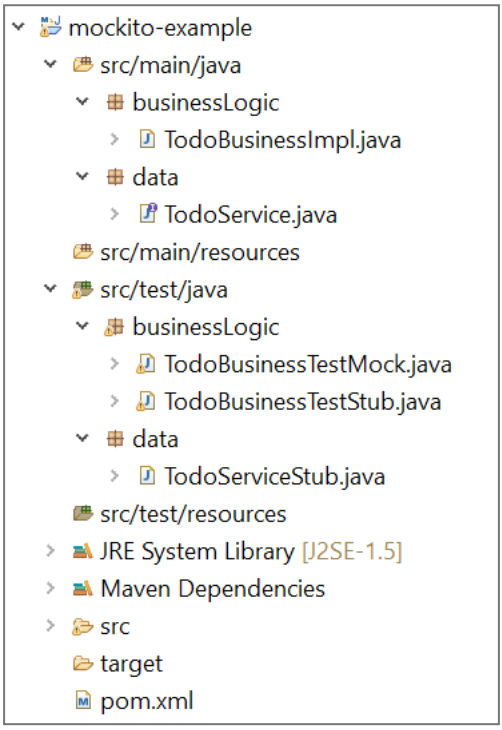

Рисунок 14. Размещение создаваемых классов в папках Maven проекта

3.4. Создание стаба

Стаб — это обычный класс, который реализует интерфейс TodoService. Его работа достаточно примитивна — метод retrieveTodos() возвращает определенный список строк.

```
// стаб - класс, который дает простую реализацию
// интерфейса TodoService для тестирования
// TodoBusinessImpl
public class TodoServiceStub implements TodoService{
```

```
    @Override
    public List<String> retrieveTodos(String user) {
        return Arrays.asList("Learn Spring MVC","Learn
                                Spring","Learn to Dance");
    }
}
```

Поскольку стаб не относится к основному коду, а служит для целей тестирования, он должен располагаться в папке src/test/java в пакете data.

Тест с использование стаба приведен ниже. Создается экземпляр стаба, который передается в конструктор объекта класса TodoBusinessImpl. Таким образом, когда будет выполняться метод retrieveTodosRelatedToSpring(), он будет использовать реализацию метода retrieveTodos(), определенную в стабе. Результат фильтрации задач — это список из двух строк. Последний оператор именно это и проверяет.

```
public class TodoBusinessImplStubTest {
  @Test
  public void testWithStub() {
    // создание заглушки
    TodoService todoServiceStub=new TodoServiceStub();
    TodoBusinessImpl todoBusinessImpl=
        new TodoBusinessImpl(todoServiceStub);
    List filteredTodos = todoBusinessImpl.
        retrieveTodosRelatedToSpring("Dummy");
    // проверка, что в отфильтрованном списке
    // два элемента
    assertEquals(2,filteredTodos.size());
  }
}
```

В чем недостаток такого подхода? Очевидно, что в отсутствии гибкости. Этот вариант метода retrieveTodos() никак не реагирует на параметр — имя пользователя. Если бы мы захотели, чтобы для разных пользователей выдавались разные списки задач, то код стал бы значительно сложнее. В итоге фактически пришлось бы написать внешний код, от которого мы хотим изолироваться.

Кроме того, в нашем простом примере интерфейс содержит только один метод, который мы должны «подменить». Если бы он содержал несколько методов, то в классе-стабе их все пришлось бы реализовать, даже если в тесте они не используются.

Таким образом, необходим более динамичный инструмент, который бы позволил легко менять сценарий работы «заглушки» с минимальными усилиями. И этот инструмент есть — это Mockito.

3.4. Простой тест с использованием Mockito

Мок создается методом mock() класса Mockito, в который передается идентификатор класса, который нужно имитировать. По умолчанию все методы мока возвращают нулевые значения. Поэтому если мок никак не настроить, то в представленном ниже тесте будет получен пустой список отфильтрованных задач:

```
public class TodoBusinessTestMock {
  @Test
  public void testWithMock() {
    // создание мока
    TodoService todoServiceMock =
                              mock(TodoService.class);
```

```
    TodoBusinessImpl todoBusinessImpl =
        new TodoBusinessImpl(todoServiceMock);

    List filteredTodos = todoBusinessImpl.
        retrieveTodosRelatedToSpring("Dummy");
    // по умолчанию мок возвращает нулевые значения
    // поэтому проверка, что в отфильтрованном
    // списке 0 элементов
    assertEquals(0,filteredTodos.size());
  }
}
```

Для настройки мока используются методы when() и thenReturn() в конструкции вида

- **when(mock.method_call()).thenReturn(value);**

Например (предполагается, что список todos уже определен):

```
when(todoServiceMock.retrieveTodos("Dummy")).
     thenReturn(todos);
```

Таким образом, приведенный ниже тест с моком работает так же, как описанный выше тест со стабом (при этом дополнительных классов создавать не пришлось):

```
public class TodoBusinessTestMock {

   @Test
   public void testWithMock() {
      // создание мока
      TodoService todoServiceMock =
                  mock(TodoService.class);
```

```
        // описание поведения мока
        List<String> todos = Arrays.asList(
                                "Learn Spring MVC",
                                "Learn Spring",
                                "Learn to Dance");
        when(todoServiceMock.retrieveTodos("Dummy")).
             thenReturn(todos);
        // создание тестируемого объекта,
        // взаимодействующего с моком
        TodoBusinessImpl todoBusinessImpl =
             new TodoBusinessImpl(todoServiceMock);
        // запуск тестируемого метода
        List filteredTodos = todoBusinessImpl.
             retrieveTodosRelatedToSpring("Dummy");
        // проверка, что в отфильтрованном списке
        // два элемента
        assertEquals(2, filteredTodos.size());
    }
}
```

3.5. Основные возможности Mockito

Рассмотрим базовые варианты настройки мока. Для примера возьмем мок, который должен подменить собой всем известный интерфейс List.

1. Настройка состоит в том, что при вызове метода size() возвращается значение 2:

```
public class ListTest {

    @Test
    public void listTest() {
        // создание мока,подменяющего List
        List listMock = mock(List.class);
```

```
        // настройка: при вызове метода size()
        // должно возвращаться 2
        when(listMock.size()).thenReturn(2);
        // вызов size() и проверка, что результат 2
        assertEquals(2,listMock.size());
    }
}
```

Следует отметить, что при такой настройке мока результат метода size() всегда будет 2, сколько бы вызовов не было. Можно указать несколько раз thenReturn() с разными значениями. Тогда при первом вызове мок возвратит первое значение, при втором — второе и т.д. При еще большем числе вызовов будет повторяться последнее указанное в thenReturn() значение.

Например, следующий тест успешно проходит:

```
@Test
public void listTestMultiple() {
    // создание мока,подменяющего List
    List listMock = mock(List.class);
    // настройка: при вызове метода size()
    // первый раз должно возвращаться 2
    // а второй и последующие разы 3
    when(listMock.size()).thenReturn(2).thenReturn(3);
    // вызов size() и проверки
    assertEquals(2,listMock.size());
    assertEquals(3,listMock.size());
    assertEquals(3,listMock.size());
}
```

2. Рассмотрим теперь имитацию работы метода с параметрами. Например, метода get() интерфейса List.

Зададим поведение мока, при котором вызов get() с аргументом 0 возвращает строку «result». При вызове этого метода с другими параметрами мок будет вести себя по умолчанию (т.е. выдавать нулевые значения). Следующий тест проходит:

```
@Test
public void listTestGet() {
    // создание мока,подменяющего List
    List listMock = mock(List.class);

    // настройка: при вызове метода get(0)
    // должна возвращаться строка "result"
    when(listMock.get(0)).thenReturn("result");

    // вызов get() и проверка, что результат "result"
    assertEquals("result",listMock.get(0));
    assertEquals(null,listMock.get(1)); // по умолчанию
                                        // результат null
}
```

В классе Matchers, входящем в состав Mockito, имеется ряд методов, которые представляют любое число определенного типа. Например, anyInt() — любое число типа int (аналогично anyDouble(), anyString() и т.п.) В тесте ниже результатом работы метода get() будет строка «result» при любом целом аргументе.

```
@Test
public void listTestGetAnyParametr() {
    // создание мока,подменяющего List
    List listMock=mock(List.class);
    // настройка: при вызове метода get() с любым целым
    // параметром должна возвращаться строка "result"
    when(listMock.get(anyInt())).thenReturn("result");
```

```
    // вызов get() и проверка, что результат "result"
    assertEquals("result",listMock.get(0));
    assertEquals("result",listMock.get(1));
}
```

3. Также можно настроить мок так, чтобы при вызове метода с заданным аргументом выбрасывалось исключение. Для этого служит метод

- **when(mock.method_call()).thenThrow(Exception_object);**

В примере ниже исключение RuntimeException выбрасывается при вызове метода get() с любым целым параметром:

```
@Test(expected=RuntimeException.class)
public void listExceptonTest() {
    // создание мока,подменяющего List
    List listMock=mock(List.class);
    // настройка: при вызове метода get() с любым целым
    // параметром должно выбрасываться исключение
    when(listMock.get(anyInt())).
         thenThrow(new RuntimeException(
         "Some message"));
    listMock.get(0);//здесь возникает исключение
}
```

3.6. Контроль вызова методов в Mockito

Контроль вызова методов — это типичная задача при использовании моков. Она особенно актуальна в том случае, когда тестируемый метод не возвращает никакого результата. Как же тогда проверить, что он выполняется корректно? Один из вариантов — проверить, сколько и ка-

кие вызовы к моку порождает этот метод во время своей работы. Можно проверить, например, что метод мока был вызван с определенным параметром или определенное число раз (либо проверить, что он не был вызван ни разу т.п.). Для такой проверки используется метод

- **verify(mock_object).method_call();**

Для демонстрации возможностей контроля вызова методов немного модифицируем наш пример про условный сервис обработки задач пользователя. Добавим в интерфейс TodoService еще один метод, который удаляет задачу из списка задач:

```
public interface TodoService {
   // метод возвращает список задач пользователя
   public List<String> retrieveTodos(String user);
   // метод удаляет задачу из списка задач
   public void deleteTodo(String todo);
}
```

В тестируемый класс TodoBusinessImpl также добавим метод, который из списка задач пользователя удаляет все задачи, не связанные со «Spring»:

```
// еще один тестируемый метод - удалить из списка
// задачи, не связанные со Spring
public void deleteTodosNotRelatedToSpring(String user) {
   List<String> todos=todoService.retrieveTodos(user);
   for(String todo:todos) {
      if(!todo.contains("Spring")) {
            todoService.deleteTodo(todo);
      }
   }
}
```

При создании теста для такого метода мы настроим мок так, чтобы его метод retrieveTodos() возвращал список из трех строк, одна из которых («Learn to Dance») не содержит подстроки «Spring». Затем запускается тестируемый метод deleteTodosNotRelatedToSpring(). Проверка состоит в том, что во время его выполнения метод заглушки deleteTodo() вызывался с параметром «Learn to Dance».

```
// тест для метода deleteTodosNotRelatedToSpring()
@Test
public void testWithDeleteTodos() {
    // Arrange - настройка условий для теста
    // создание заглушки
    TodoService todoServiceMock=mock(TodoService.class);
    // настройка мока
    List<String> todos=Arrays.asList("Learn Spring MVC",
                                     "Learn Spring",
                                     "Learn to Dance");
    when(todoServiceMock.retrieveTodos("Dummy")).
         thenReturn(todos);
    // создание тестируемого объекта
    TodoBusinessImpl todoBusinessImpl =
         new TodoBusinessImpl(todoServiceMock);
    // Act - запуск метода
    todoBusinessImpl.
         deleteTodosNotRelatedToSpring("Dummy");
    // Assert - проверка, что метод мока deleteTodo()
    // вызывался с параметром "Learn to Dance"
    verify(todoServiceMock).
         deleteTodo("Learn to Dance");
}
```

Аналогично можно проверить, что метод НЕ будет вызван с определенным параметром, если добавить в качестве второго аргумента verify() метод never():

```
// проверка, что метод не будет вызван с параметром
// Learn Spring
verify(todoServiceMock, never()).
       deleteTodo("Learn Spring");
```

Если же использовать метод times(), то можно проверить, сколько раз вызывался метод с данным параметром

```
// проверка, что метод с данным параметром
// вызывается один раз
verify(todoServiceMock,times(1)).
       deleteTodo("Learn to Dance");
```

Еще варианты второго аргумента verify():

```
// проверка, что метод вызывается по крайней мере
// один раз
verify(todoServiceMock,atLeastOnce()).
       deleteTodo("Learn to Dance");
// другой вариант того же
verify(todoServiceMock,atLeast(1)).
       deleteTodo("Learn to Dance");
```

3.7. Захват аргументов, передаваемых в метод мока

При тестировании часто бывает необходимо выяснить, с каким именно аргументом вызывается метод мока. Для определения этого используется механизм «захвата» аргумента и класс ArgumentCaptor. Этот класс должен параметризован типом аргумента. В нашем примере это тип String, но в реальных проектах это может быть более сложный тип данных.

В тесте нужно выполнить три шага при использовании «захватчика аргументов»:

1. Объявить ссылочную переменную и создать объект класса AgrumentCaptor.
2. Использовать метод capture() созданного объекта вместо аргумента исследуемого метода мока в описании verify(mock).method_call()
3. Получить значение аргумента от объекта класса AgrumentCaptor, используя его метод экземпляра getValue() (либо getAllValues() при множественном захвате аргументов).

Пример:

```
// тест  с захватом аргумента для метода
// deleteTodosNotRelatedToSpring()
@Test
public void testWithDeleteTodos2() {
   // Объявить "захватчика" аргумента
ArgumentCaptor<String> stringArgCaptor =
              ArgumentCaptor.forClass(String.class);
   // создание и настройка мока
   TodoService todoServiceMock =
              mock(TodoService.class);
   List<String> todos = Arrays.asList("Learn Spring MVC",
                                      "Learn Spring",
                                      "Learn to Dance");
   when(todoServiceMock.retrieveTodos("Dummy")).
              thenReturn(todos);
   TodoBusinessImpl todoBusinessImpl =
              new TodoBusinessImpl(todoServiceMock);
   // Запуск тестируемого метода
   todoBusinessImpl.
              deleteTodosNotRelatedToSpring("Dummy");
```

```
    // захват аргумента в вызываемом методе мока
    verify(todoServiceMock).deleteTodo(stringArgCaptor.
            capture());
    // проверка, что захваченный аргумент -
    // это строка "Learn to Dance"
    assertThat(stringArgCaptor.getValue(),
                is("Learn to Dance"))
}
```

А в следующем примере проверяется, что метод deleteTodo() вызывается два раза:

```
// тест  с захватом нескольких аргументов
// для метода deleteTodosNotRelatedToSpring()
@Test
public void testWithDeleteTodos3() {
    // Объявить "захватчика" аргумента
    ArgumentCaptor<String> stringArgCaptor =
            ArgumentCaptor.forClass(String.class);
    // создание и настройка мока
    TodoService todoServiceMock =
            mock(TodoService.class);
    List<String> todos =
            Arrays.asList("Learn to Rock and Roll",
                          "Learn Spring",
                          "Learn to Dance");
    when(todoServiceMock.retrieveTodos("Dummy")).
            thenReturn(todos);
    TodoBusinessImpl todoBusinessImpl =
            new TodoBusinessImpl(todoServiceMock);
    // Запуск тестируемого метода
    todoBusinessImpl.
            deleteTodosNotRelatedToSpring("Dummy");
    // множественный захват аргумента в вызываемом
    // методе мока
```

```
verify(todoServiceMock,times(2)).
        deleteTodo(stringArgCaptor.capture());
    // проверка количества захваченных аргументов
    assertThat(stringArgCaptor.getAllValues().
        size(),is(2));
}
```

Отметим, что при использовании Mockito мы получаем вполне исчерпывающее объяснение ситуации в случае, когда тест не проходит. Например, для теста ниже мы получим комментарии в окне JUnit, которые показаны на рисунке 15.

```
// тест для метода deleteTodosNotRelatedToSpring()
@Test
public void testWithDeleteTodos() {

    // Arrange - настройка условий для теста
    TodoService todoServiceMock=mock(TodoService.class);
    List<String> todos =
        Arrays.asList("Learn Spring MVC","Learn Spring",
                      "Learn to Dance");
    when(todoServiceMock.retrieveTodos("Dummy")).
         thenReturn(todos);
    TodoBusinessImpl todoBusinessImpl =
         new TodoBusinessImpl(todoServiceMock);

    // Act - запуск метода
    todoBusinessImpl.
         deleteTodosNotRelatedToSpring("Dummy");

    // Assert - проверка, что метод мока deleteTodo
    // вызывался с параметром "Learn Spring"
    verify(todoServiceMock).deleteTodo("Learn Spring");
}
```

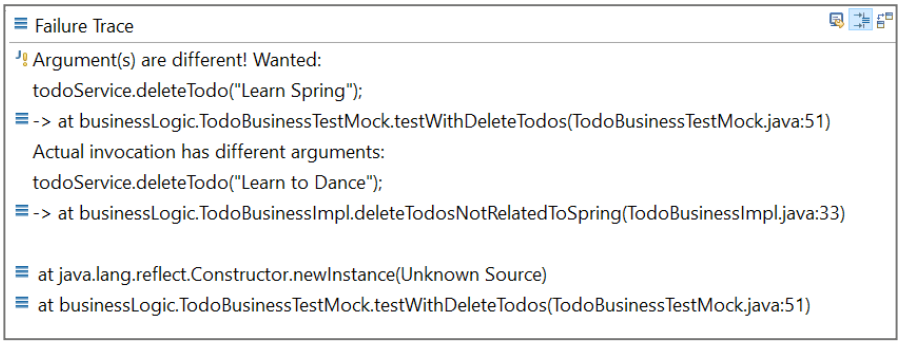

Рисунок 15. Информация о не прошедшем тесте в окне JUnit

Фреймворк Mockito имеет множество других замечательных возможностей, изучить которые можно здесь. Хороший видеокурс по Mockito от O'Reilly можно найти тут.