Использование JUnit и Mockito

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

- В зависимости от анализируемых аспектов кода различают следующие виды тестирования:

- 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.

- 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).

- 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.

- 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:

- 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.

- 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

- 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

- Тестирование «черного ящика» — тестировщик не знает, как устроен код, а создает набор тестов только на основе спецификации к программе.
- 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.

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.


начало
Написать тест
Проходит?
да
нет
Написать код
Запустить все тесты
Все тесты проходят?
нет
да
Рефакторить код

- 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

Инструменты модульного тестирования на Java:

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

- 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.

JUnit

- JUnit — это один из наиболее популярных фреймворков для разработки юнит-тестов (модульного тестирования) на Java

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:

```java
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.

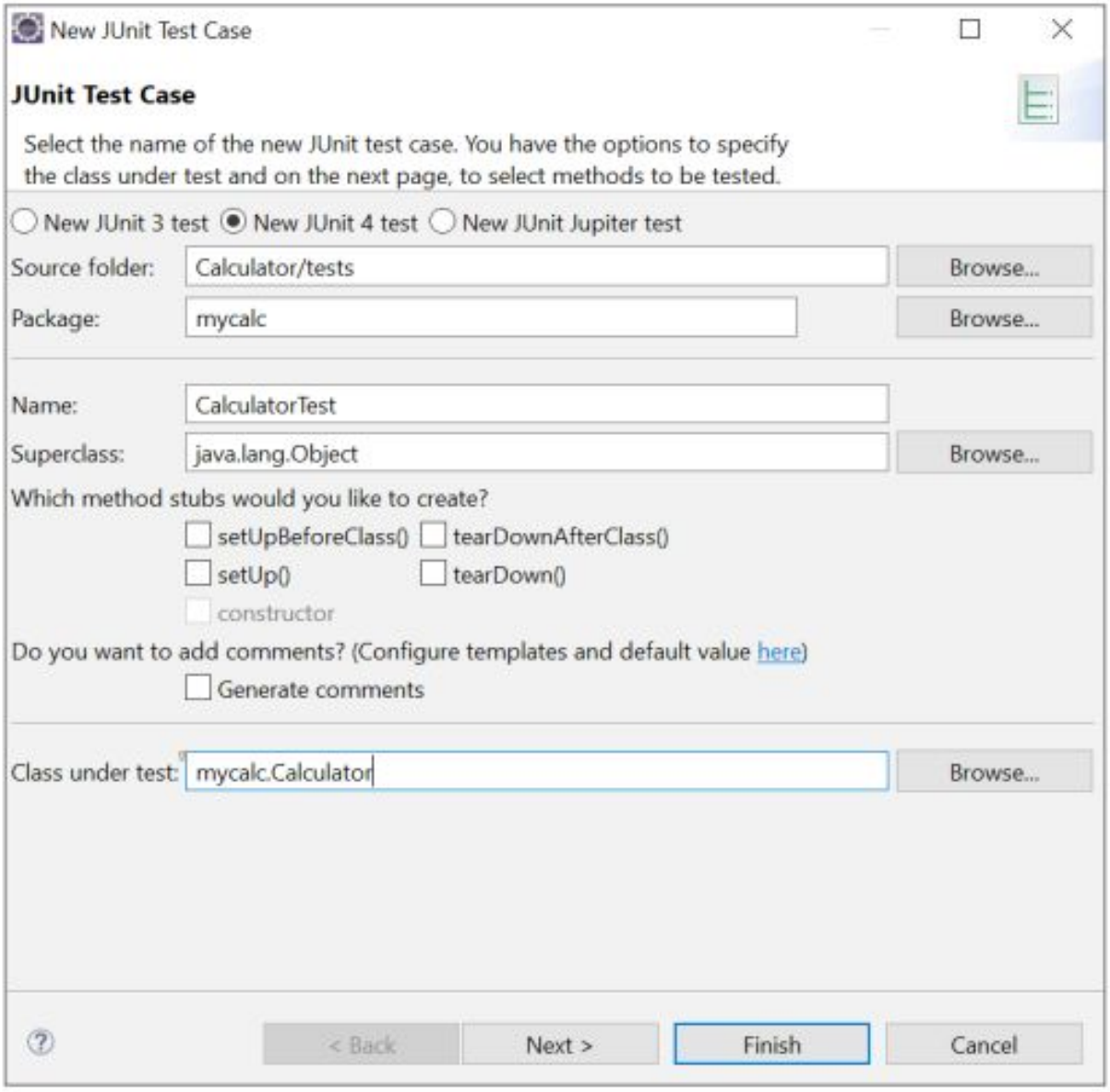

Рисунок 2. Окно создания тестового класса

- 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

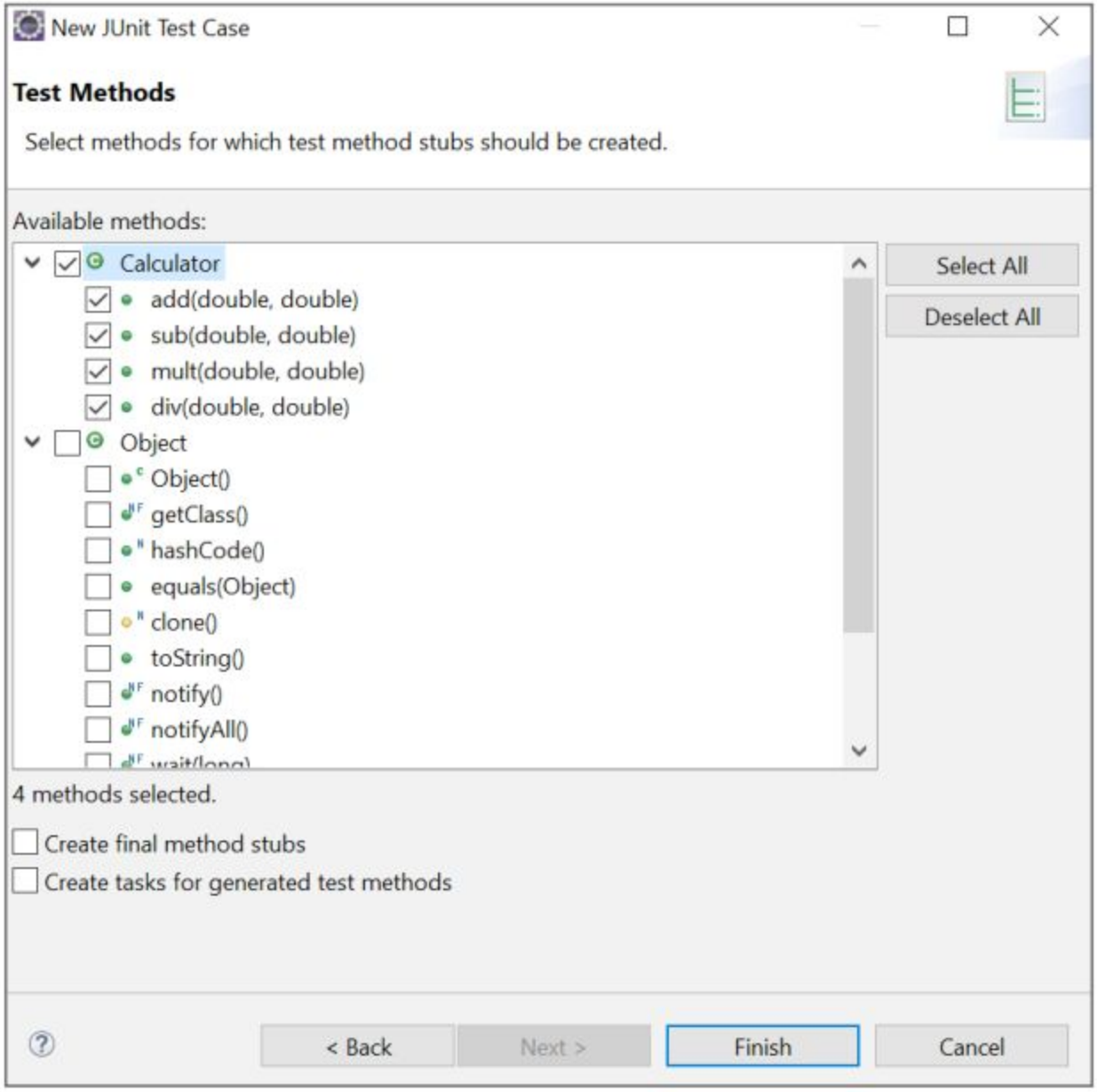

Рисунок 3. Выбор тестируемых методов

- 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.

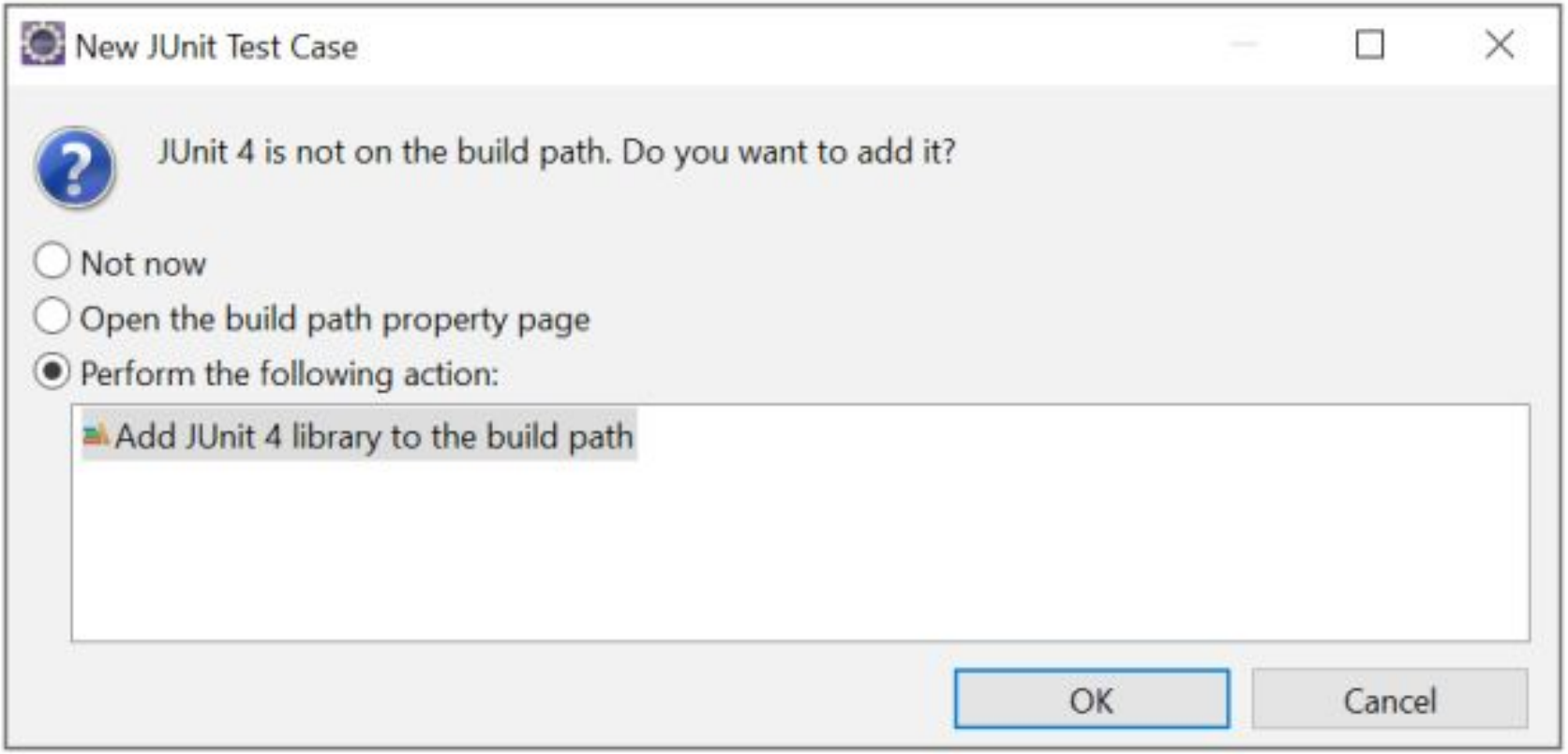

Рисунок 4. Добавление фреймворка JUnit в проект

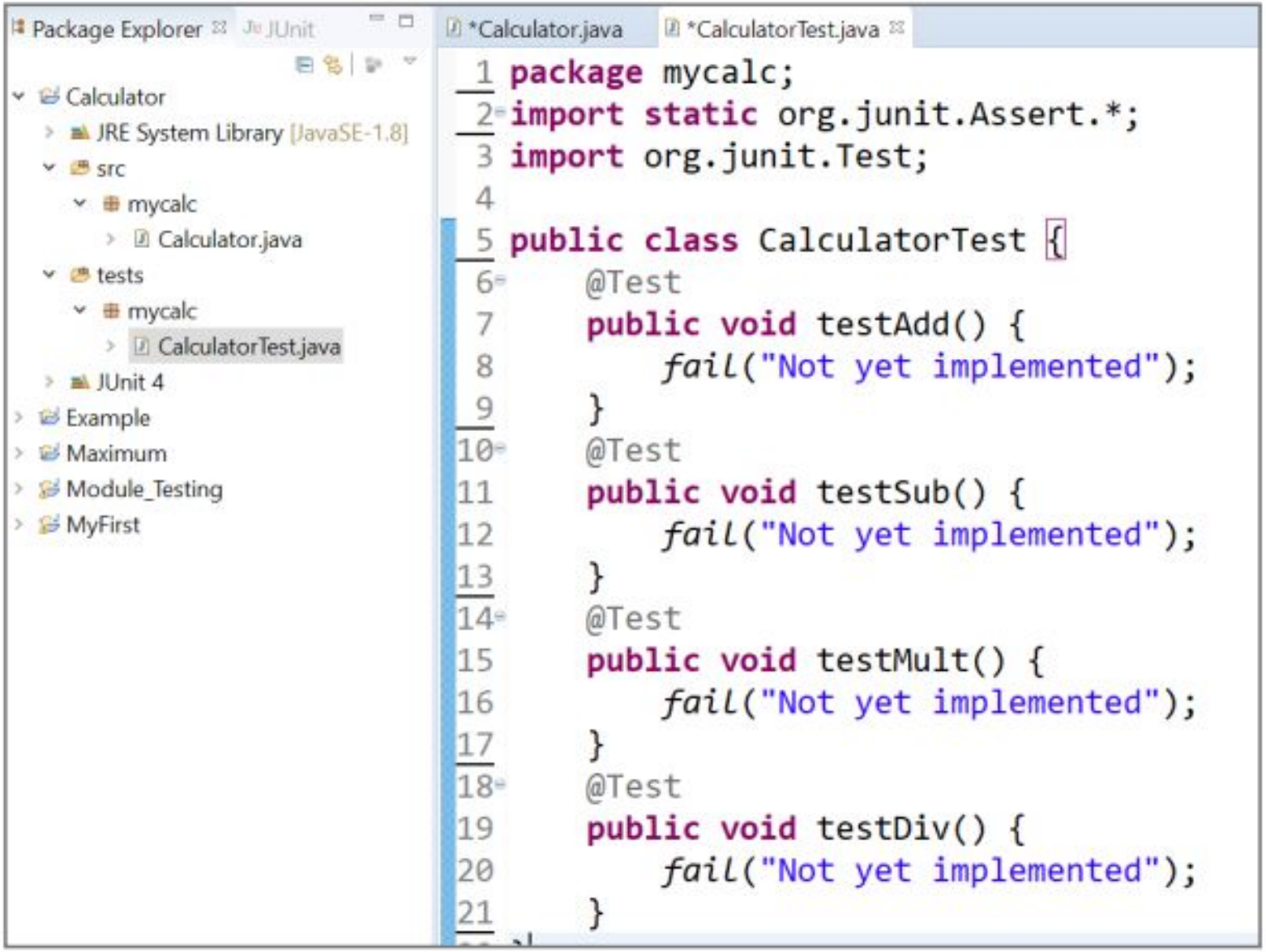

Рисунок 5. Заготовка тестирующего класса после создания

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:

- 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

- 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 завaленным (failed).

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

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.

- 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:

```java
@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
}
```

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.

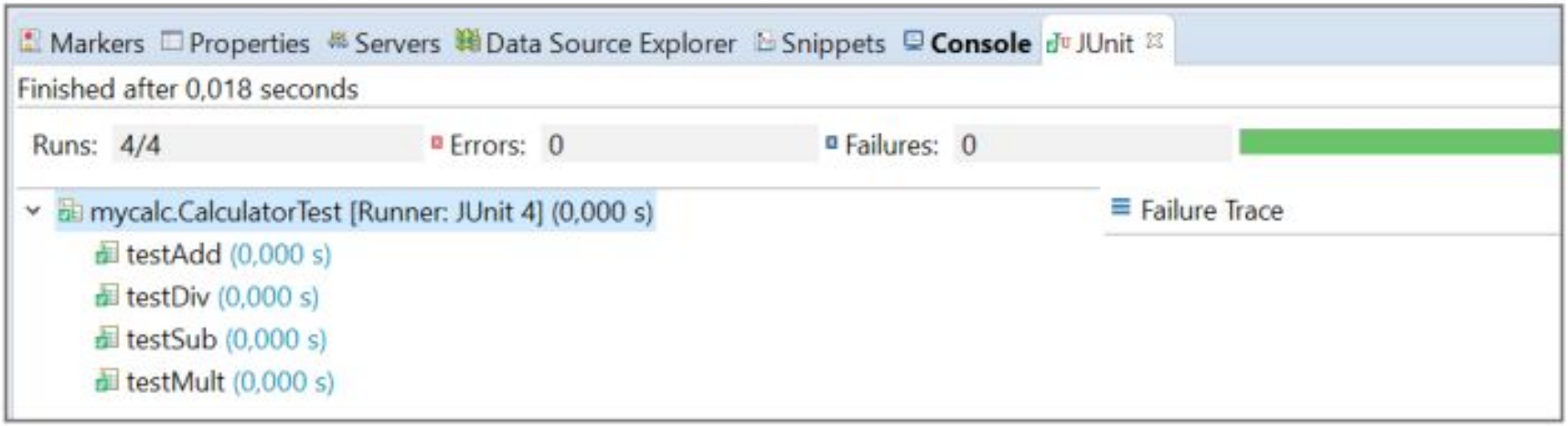

Рисунок 6. Окно результатов модульного тестирования

Фикстуры

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

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 возникло, 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.

Тестирование времени выполнения метода

- 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);
    }
}
```

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.

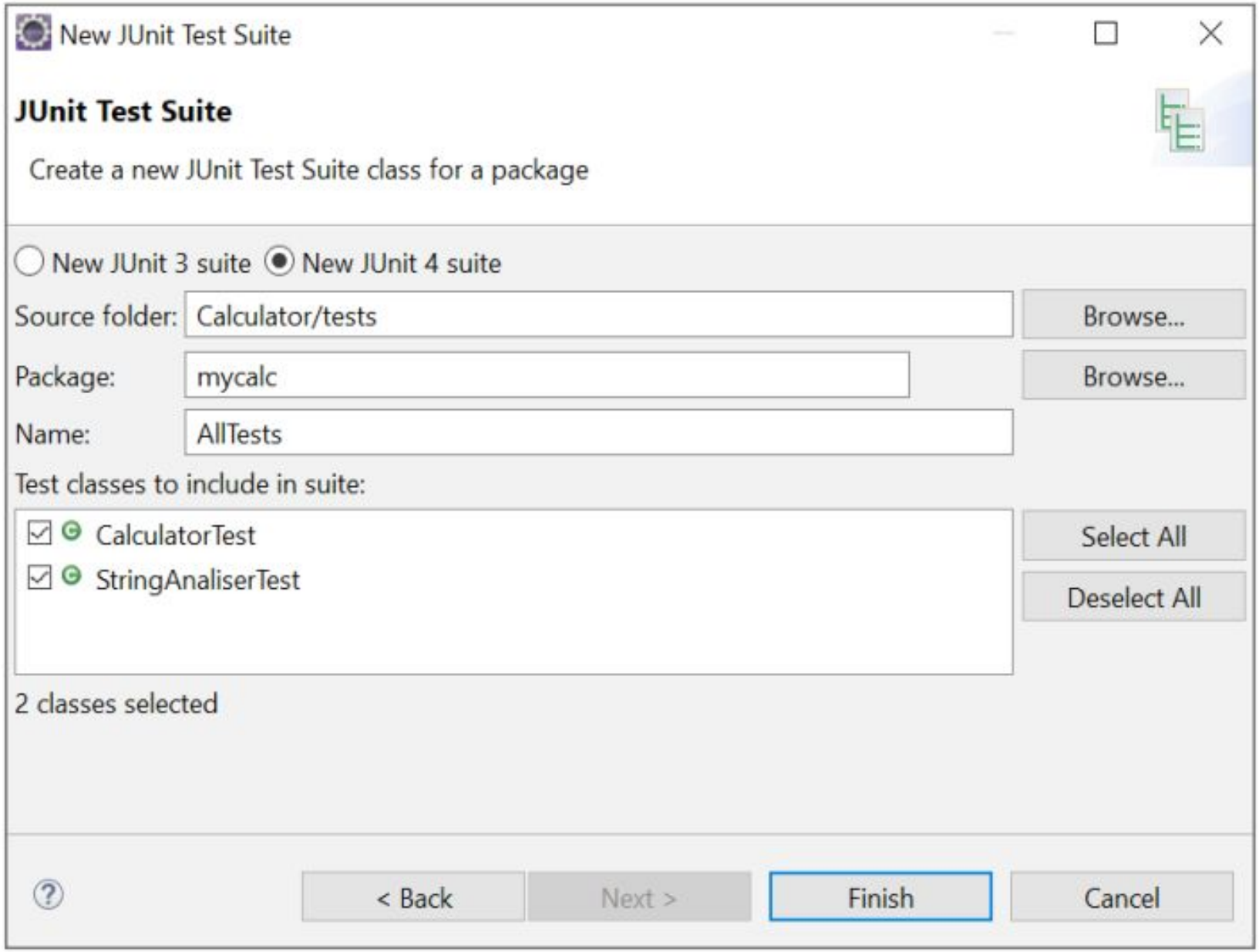

Рисунок 7. Окно создания набора тестов

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 {
}
```