

JUnit

Оболочки модульного тестирования — это программные средства для разработки тестов, включающие: построение, выполнение тестов и создание отчетов. Первая оболочка модульного тестирования SUnit была создана Кентом Бекем в 1999 году для языка Smalltalk. Позднее Эрик Гамма создал JUnit. Сейчас существует множество оболочек для модульного тестирования, которые известны как программные средства семейства XUnit. JUnit — реализация XUnit, наиболее широко используемая и расширенная версия оболочек модульного тестирования. JUnit создан на языке Java и используется для тестирования Java кода.

Модульное тестирование или *unit testing* — процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде. Под частью кода в Java следует понимать исполняемый компонент. С помощью модульного тестирования обычно тестируют низкоуровневые элементы кода — такие, как методы. JUnit позволяет вне исследуемого класса создавать тесты, при выполнении которых произойдет корректное завершение программы. Кроме основного положительного сценария может выполняться проверка работоспособности системы в альтернативных сценариях, например, при генерации методом исключения как реакция на ошибочные исходные данные.

Оценивая каждую часть кода изолированно и подтверждая корректность ее работы, точно установить проблему значительно проще, чем если бы элемент был частью системы.

Технология позволяет и предлагает сделать более тесной связь между разработкой кода и его тестированием, а также предоставляет возможность проверить корректность работы класса, не прибегая к пробному выводу при отладке кода.

JUnit 4 в отличие от JUnit 3 полностью построен на аннотациях.

Для использования технологии необходимо загрузить библиотеку JUnit с сервера junit.org и включить архив [junit.jar](#) в список библиотек приложения.

При включении модульного тестирования в проект:

- тесты разрабатываются для нетривиальных методов системы;
- ошибки выявляются в процессе проектирования метода или класса;
- в первую очередь разрабатываются тесты на основной положительный сценарий;
- разработчику приходится больше уделять внимания альтернативным сценариям поведения, так как они являются источником ошибок, выявляемых на поздних стадиях разработки;

- разработчику приходится создавать более сфокусированные на своих обязанностях методы и классы, так как сложный код тестировать значительно труднее;
- снижается число новых ошибок при добавлении новой функциональности;
- устаревшие тесты можно игнорировать;
- тест отражает элементы технического задания, то есть некорректное завершение теста сообщает о нарушении технических требований заказчика;
- каждому техническому требованию соответствует тест;
- получение работоспособного кода с наименьшими затратами.

Аннотация @Test

Аннотация помечает метод как тестовый, что позволяет использовать возможности класса **org.junit.Assert** и запускать его в режиме тестирования. Тестовый метод должен всегда объявляться как **public void**. Аннотация может использовать параметры: **expected** — определяет ожидаемый класс исключения; **timeout** — определяет время, превышение которого делает тест ошибочным, применение которых будет рассмотрено ниже.

Пусть необходимо создать тест на метод, производящий простые вычисления студенческой стипендии.

```
/* # 1 # расчет стипендии в зависимости от повышающего коэффициента #
IScholarshipCalculator.java # ScholarshipCalculatorImpl.java */
```

```
package by.bsu.calculation;
public interface IScholarshipCalculator {
    public double scholarshipCalculate(double stepUpCoefficient);
}
package by.bsu.calculation;
public class ScholarshipCalculatorImpl implements IScholarshipCalculator{
    public static final double BASIC_SCHOLARSHIP = 100;
    public double scholarshipCalculate(double stepUpCoefficient) {
        return BASIC_SCHOLARSHIP * stepUpCoefficient;
    }
}
```

Метод, предназначенный для функционирования в качестве теста, достаточно промаркировать аннотацией **@Test**. Простейший тест на метод будет выглядеть следующим образом.

```
/* # 2 # тест с аннотацией @Test # ScholarshipCalculatorTest.java */
```

```
package test.by.bsu.calculation;
import org.junit.Assert;
import org.junit.Test;
import by.bsu.calculation.IScholarshipCalculator;
import by.bsu.calculation.ScholarshipCalculatorImpl;
```

```

public class ScholarshipCalculatorTest {
    @Test
    public void scholarshipCalculateTest() {
        IScholarshipCalculator scholarshipCalculator =
            new ScholarshipCalculatorImpl();
        double basicScholarship = ScholarshipCalculatorImpl.BASIC_SCHOLARSHIP;
        double stepUpCoefficient = 1.1;
        double expected = basicScholarship * stepUpCoefficient;
        double actual = scholarshipCalculator
            .scholarshipCalculate(stepUpCoefficient);
        // проверка на совпадение с погрешностью 0,01
        Assert.assertEquals(expected, actual, 0.01);
        // Assert.assertEquals( "Тест не прошел, т.к.", expected, actual, 0.01);
        // устаревшие варианты :
        // Assert.assertEquals(expected, actual); // на точное совпадение – deprecated
        // Assert.assertEquals( "Тест не прошел, т.к.", expected, actual); // deprecated
    }
}

```

Метод **assertEquals()** проверяет на равенство значений **expected** и **actual** с возможной погрешностью **delta**. При выполнении заданных условий сообщает об успешном завершении, в противном случае — об аварийном завершении теста. При аварийном завершении генерируется ошибка **java.lang.AssertionError**. Все методы класса **Assert** в качестве возвращаемого значения имеют тип **void**. Среди них можно выделить:

assertTrue(boolean condition)/assertFalse(boolean condition) — проверяет на истину/ложь значение **condition**;

assertSame(Object expected, Object actual) — проверяет, ссылаются ли ссылки на один и тот же объект;

assertNotSame(Object unexpected, Object actual) — проверяет, ссылаются ли ссылки на различные объекты;

assertNull(Object object)/assertNotNull(Object object) — проверяет, имеет или не имеет ссылка значение **null**;

assertThat(T actual, Matcher<T> matcher) — проверяет выполнение условия;

fail() — вызывает ошибку, используется для проверки, достигнута ли определенная часть кода или для заглушки, сообщающей, что тестовый метод пока не реализован.

Все перечисленные методы имеют перегруженную версию с параметром типа **String** в первой позиции, в который можно передавать сообщение, выводимое при аварийном завершении теста.

В дополнение к классу **Assert** с его методами жесткой проверки прохождения теста разработан класс **org.junit.Assume**. Методы этого класса в случае невыполнения предполагаемого условия при работе теста сообщают только о том, что предположение не исполнилось, не генерируя при этом никаких ошибок. Методы класса предполагают, что:

assumeNoException(Throwable t) — тестируемый метод завершится, не вызвав исключения;

assumeNotNull(Object...objects) — передаваемый аргумент(ы) не является ссылкой на **null**;

assumeThat(T actual, Matcher<T> matcher) — условие выполнится;

assumeTrue(boolean b) — значение передаваемого аргумента истинно.

Фикстуры

Фикстура (Fixture) — состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и проч.

В версии JUnit 4 аннотации позволяют исполнять одну и ту же фикстуру для каждого теста или всего один раз для всего класса, или не исполнять ее совсем. Предусмотрено четыре аннотации фикстур — две для фикстур уровня класса и две для фикстур уровня метода.

— **@BeforeClass** — запускается только один раз при запуске теста.

— **@Before** — запускается перед каждым тестовым методом.

— **@After** — запускается после каждого метода.

— **@AfterClass** — запускается после того, как отработали все тестовые методы.

Использование фикстур позволяет выделить этапы и точно определять моменты, например, создания/удаления объекта, инициализации необходимых ресурсов, очистки памяти и проч.

Пусть метод **stepUpCoefficientCalculate(int averageMark)** возвращает повышающий коэффициент стипендии в зависимости от среднего балла студента.

```

/* # 3 # тестируемый класс # ScholarshipCalculatorImpl.java */

package by.bsu.calculation;
public class ScholarshipCalculatorImpl implements IScholarshipCalculator {
    public static final double BASIC_SCHOLARSHIP = 100;
    public double scholarshipCalculate(double stepUpCoefficient) {
        return BASIC_SCHOLARSHIP * stepUpCoefficient;
    }
    public double stepUpCoefficientCalculate(int averageMark) {
        double stepUpCoefficient;
        switch (averageMark) {
            case 3:
                stepUpCoefficient = 1;
                break;
            case 4:
                stepUpCoefficient = 1.3;
                break;
        }
    }
}

```

```

        case 5:
            stepUpCoefficient = 1.5;
            break;
        default:
            stepUpCoefficient = 0;
    }
    return stepUpCoefficient;
}
}

```

Фикстурой **@Before** задан момент создания объекта класса перед каждым тестом.

```
/* # 4 # тест с фикстурами @Before и @After # ScholarshipCalculatorTest2.java */
```

```

package test.by.bsu.calculation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTest2 {
    private ScholarshipCalculatorImpl scholarshipCalculator;
    @Before
    public void initScholarshipCalculator() {
        scholarshipCalculator = new ScholarshipCalculatorImpl();
    }
    @After
    public void clearScholarshipCalculator() {
        scholarshipCalculator = null;
    }
    @Test
    public void stepUpCoefficientForFiveTest() {
        double expected = 1.5;
        double actual = scholarshipCalculator.stepUpCoefficientCalculate(5);
        assertEquals("Coefficient for mark 5 is wrong:", expected, actual, 0.01);
    }
    @Test
    public void stepUpCoefficientForThreeTest() {
        double expected = 1;
        double actual = scholarshipCalculator.stepUpCoefficientCalculate(3);
        assertEquals("Coefficient for mark 3 is wrong:", expected, actual, 0.01);
    }
}
}

```

Метод **initScholarshipCalculator()** выполнит инициализацию поля **scholarshipCalculator** перед запуском каждого тестового метода. Метод **clearScholarshipCalculator()** очистит это поле после завершения работы каждого из тестовых методов.

Тест можно переписать с использованием фикстуры **@BeforeClass**, чье использование позволит создавать только один экземпляр класса, на котором и будут выполнены оба теста.

```

/* # 5 # тест с фикстурой @BeforeClass # ScholarshipCalculatorTest3.java */

package test.by.bsu.calculation;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTest3 {
    private static ScholarshipCalculatorImpl scholarshipCalculator;
    @BeforeClass
    public static void initScholarshipCalculator() { // static обязателен
        scholarshipCalculator = new ScholarshipCalculatorImpl();
    }
    @Test
    public void stepUpCoefficientForFiveTest() {
        double expected = 1.5;
        double actual = scholarshipCalculator.stepUpCoefficientCalculate(5);
        assertEquals("Coefficient for mark 5 is wrong:", expected, actual, 0.01);
    }
    @Test
    public void stepUpCoefficientForThreeTest() {
        double expected = 1;
        double actual=scholarshipCalculator.stepUpCoefficientCalculate(3);
        assertEquals("Coefficient for mark 3 is wrong:", expected, actual , 0.01);
    }
}

```

Метод **initScholarshipCalculator()** отрабатывает один раз перед вызовом первого тестового метода. Метод, отмеченный аннотацией **@BeforeClass** или **@AfterClass**, должен быть статическим.

JUnit 4 позволяет указать для тестового сценария более одной фикстуры. Новые фикстуры на основе аннотаций не препятствуют созданию нескольких фикстурных методов **@BeforeClass**. Однако их порядок исполнения задать нельзя.

Тестирование исключительных ситуаций

При тестировании альтернативных сценариев работы метода часто требуется точно определить тип генерируемого методом исключения на основе переданных некорректных параметров. Если тест выдает исключение, то инфраструктура тестирования сообщает о корректном результате его исполнения.

Аннотацию **@Test** при необходимости тестирования генерации конкретного исключения следует использовать с параметром **expected**. Параметр предназначен

для задания типа исключения, которое данный тест должен генерировать в процессе своего выполнения.

```
/* # 6 # метод с инструкцией throws # */
```

```
public double stepUpCoefficientCalculate(int averageMark) throws NotSuchMarkException {
    double stepUpCoefficient;
    switch (averageMark) {
        case 2:    stepUpCoefficient = 0;
                 break;
        case 3:
                 stepUpCoefficient = 1;
                 break;
        case 4:
                 stepUpCoefficient = 1.3;
                 break;
        case 5:
                 stepUpCoefficient = 1.5;
                 break;
        default:
                 throw new NotSuchMarkException("There is no mark: " + averageMark);
    }
    return stepUpCoefficient;
}
```

Метод **stepUpCoefficientCalculate(int averageMark)** генерирует исключение **NotSuchMarkException**, если число, переданное в метод, не входит в интервал возможных оценок.

```
/* # 7 # исключение # NotSuchMarkException.java */
```

```
package by.bsu.calculation;
public class NotSuchMarkException extends Exception {
    public NotSuchMarkException() {
    }
    public NotSuchMarkException(String message) {
        super(message);
    }
}
```

Тогда метод, тестирующий генерацию исключения, будет записан в виде:

```
/* # 8 # метод тестирования генерации исключения # */
```

```
@Test( expected = NotSuchMarkException.class )
public void stepUpCoefficientForElevenTest() throws NotSuchMarkException {
    double expected = 1;
    double actual = scholarshipCalculator.stepUpCoefficientCalculate(11);
    Assert.assertEquals("For mark 11 wasn't exception:", expected, actual, 0.01);
}
```

Тест завершится успешно лишь в том случае, если возникнет исключительная ситуация. Исключение **NotSuchMarkException** необходимо указать в секции **throws** тестового метода по правилам работы с проверяемыми исключениями.

Может возникнуть необходимость проверить не только возникновение исключительной ситуации, но и текст сообщения в экземпляре исключения. В этом случае лучше прибегнуть к обычному подходу без параметра **expected**.

```
/* # 9 # метод тестирования сообщения в исключении после его генерации # */

@Test
public void stepUpCoefficientForElevenTest() {
    int averageMark = 11;
    try {
        scholarshipCalculator.stepUpCoefficientCalculate(averageMark);
        fail("Test for mark " + averageMark
            + " should have thrown a NotSuchMarkException");
    } catch (NotSuchMarkException e) {
        Assert.assertEquals("There is no mark: " + averageMark, e.getMessage());
    }
}
```

В блоке **try**, если исключение не возникло, вызывается метод **fail()**, сигнализирующий о провале теста. В блоке **catch**, если исключительная ситуация произошла, проверяется на эквивалентность текстов сообщения об ошибке.

Ограничение по времени

В качестве параметра тестового сценария аннотации **@Test** может быть использовано значение лимита времени **timeout**. Параметр **timeout** определяет максимальный временной промежуток в миллисекундах, отводимый на исполнение теста. Если выделенное время истекло, а тест продолжает выполняться, то тест завершается неудачей.

```
/* # 10 # тест с ограничением по времени # ScholarshipCalculatorTestTime.java */

package test.by.bsu.calculation;
import org.junit.Assert;
import org.junit.Test;
import by.bsu.calculation.IScholarshipCalculator;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTestTime {
    private static ScholarshipCalculatorImpl scholarshipCalculator;
    @BeforeClass
    public static void initScholarshipCalculator() { // static обязателен
        scholarshipCalculator = new ScholarshipCalculatorImpl();
    }
    @Test( timeout = 10 ) // заменить на 50
    public void scholarshipCalculateTest() {
```



```

        for (int i = 1; i < 100_000; i++) {
            double stepUpCoefficient = 1 / i;
            double expected = 100 * stepUpCoefficient;
            double actual =
                scholarshipCalculator.scholarshipCalculate(stepUpCoefficient);
            Assert.assertEquals(expected, actual, 0.01);
        }
    }
}

```

Через 10 миллисекунд после запуска тест провалится, так как на выполнение метода уходит несколько больше времени. Если увеличить время до 50 миллисекунд, то тест пройдет успешно.

Игнорирование тестов

При контроле корректности функционирования бизнес-логики приложений п до появления JUnit 4 игнорирование неудачных, незавершенных или устаревших тестов представляло определенную проблему. Аннотация **@Ignore** заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Аннотация предусматривает наличие комментария о причине игнорирования теста, полезного при следующем к нему обращении.

```
/* # 11 # игнорирование теста # */
```

```

@Ignore("this test is not ready yet")
@Test
public void scholarshipCalculateTest() {
    double expected = 10 * stepUpCoefficient;
    double actual = scholarshipCalculator.scholarshipCalculate(stepUpCoefficient);
    Assert.assertEquals(expected, actual, 0.1);
}

```

Правила

Аннотация **@Rule** позволяет более гибко работать с классами утилитами, определяющими правила работы с тестами. Некоторые классы пакета **org.junit.rules** повторяют функциональность параметров аннотации **@Test**, а именно, класс **Timeout** определяет таймаут для теста, класс **ExpectedException** — ожидаемое исключение. С другой стороны, класс **TemporaryFolder** дает возможность протестировать возможность управления временными файлами и директориями.

Пусть в класс **ScholarshipCalculatorImpl** добавлена возможность печати отчета в файл в методе **writeResult(File f)**. Тогда с использованием правил можно создать тест для метода записи в файл и два теста, проверяющих генерацию исключений.

```

/* # 12 # тест с правилами # ScholarshipCalculatorTestRule.java */

package test.by.bsu.calculation;
import java.io.File;
import java.io.IOException;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import org.junit.rules.TemporaryFolder;
import org.junit.rules.Timeout;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTestRule {
    private static ScholarshipCalculatorImpl scholarshipCalculator;
    @Rule
    public final TemporaryFolder folder = new TemporaryFolder();
    @Rule
    public final Timeout timeout = new Timeout(100);
    @Rule
    public final ExpectedException thrown = ExpectedException.none();
    @BeforeClass
    public static void initScholarshipCalculator() {
        scholarshipCalculator = new ScholarshipCalculatorImpl();
    }
    @Test
    public void writeResultTest() throws IOException {
        File file = folder.newFile("result.txt");
        scholarshipCalculator.writeResult(file);
    }
    @Test
    public void stepUpCoefficientCalculateTest() throws NotSuchMarkException {
        thrown.expect(NotSuchMarkException.class);
        scholarshipCalculator.stepUpCoefficientCalculate(11);
    }
    @Test
    public void writeResultTestTwo() throws IOException {
        thrown.expect(NullPointerException.class);
        scholarshipCalculator.writeResult(null);
    }
}

```

В методе **expect()** класса **ExpectedException** следует указать класс ожидаемого исключения.

Временные файлы по окончании теста уничтожаются.

Метод **writeResult()** класса **ScholarshipCalculatorImpl** представлен в виде:

```

public void writeResult(File f) throws IOException {
    FileWriter fw = new FileWriter(f);
    fw.append(this.toString());
    fw.flush();
    fw.close();
}

```

Наборы тестов и параметризованные тесты

При контроле корректности функционирования бизнес-логики приложений приходится создавать тесты, число которых достаточно велико и непостоянно. Аннотация **@RunWith** задает способ запуска теста, в случае **@RunWith(Suite.class)** — запуск набора тестов, перечисляемых в аннотации **@Suite.SuiteClasses**.

```
/* # 13 # запуск набора тестов # ScholarshipSuite.java */
package test.by.bsu.calculation;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@Suite.SuiteClasses( { ScholarshipCalculatorTest.class, ScholarshipCalculatorTestTime.class } )
@RunWith(Suite.class)
public class ScholarshipSuite {
}
```

Наличие класса после описания способа группы тестов в общем случае — необходимая формальность.

JUnit 4 позволяет создавать тест, который может работать с различными наборами значений параметров, что позволяет разработать единый тестовый сценарий и запускать его несколько раз — по числу наборов параметров. Запуск параметризованного теста с набором данных также требует аннотации **@RunWith(Parameterized.class)**, где в качестве значения передается указание на способ запуска. У класса должны быть поля по числу параметров в каждом наборе. Аннотация **@Parameters** маркирует статический метод, который возвращает данные для теста, представленные типом **Collection**. Кроме того, необходим конструктор, связывающий данные для теста и поля класса.

```
/* # 14 # запуск параметризованного набора для теста # ScholarshipCalculatorTest4.java */
package test.by.bsu.calculation;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
@RunWith(Parameterized.class)
public class ScholarshipCalculatorTest4 {
    // объявление параметров в виде полей
    private int averageMark;
    private double stepUpCoefficient;
    // public-конструктор с параметрами для инициализации полей
    public ScholarshipCalculatorTest(int averageMark, double stepUpCoefficient) {
        this.averageMark = averageMark;
        this.stepUpCoefficient = stepUpCoefficient;
    }
}
```

```

// определение набора параметров в виде коллекции
@Parameters public static Collection<Object[]> stepUpCoefficientTableValues() {
    return Arrays.asList(new Object[][] {
        { 3, 1.0 },
        { 4, 1.3 },
        { 5, 1.5 }
    });
}
@Test
public void stepUpCoefficientTest() throws NotSuchMarkException {
    ScholarshipCalculatorImpl scholarshipCalculator =
        new ScholarshipCalculatorImpl();
    double expected = this.stepUpCoefficient;
    double actual =
        scholarshipCalculator.stepUpCoefficientCalculate(this.averageMark);
    assertEquals(expected, actual, 0.01);
}
}

```

Тест будет запущен по числу наборов данных, в данном случае — четыре раза.

События

При выполнении больших или сложных тестов бывает недостаточно информации от JUnit о процессе их выполнения. Возникает необходимость документирования результатов работы или выполнения определенных программных действий. Для реализации таких возможностей применяется модель прослушивания событий.

Чтобы реализовать прослушивание событий, следует создать подкласс класса **org.junit.runner.notification.RunListener** и зарегистрировать обработчик событий в блоке прослушивания с помощью класса-регистратора наследника класса **org.junit.runners.BlockJUnit4ClassRunner**. Класс, содержащий тесты и заинтересованный в обработке событий, необходимо пометить аннотацией **@RunWith** с указанием класса регистратора событий.

Класс **RunListener** позволяет переопределять следующие методы:

testStarted(Description description) — вызывается перед запуском каждого теста. В параметр **description** передается имя метода и класса, запустившего тест;

testFinished(Description description) — вызывается после успешного завершения каждого теста;

testIgnored(Description description) — вызывается, если тест не был запущен, как правило, из-за аннотации **@Ignore**;

testFailure(Failure failure) — вызывается при неудачном завершении теста. Параметр **failure** содержит информацию о неудачном тесте и об исключении, сгенерированном в итоге;

testAssumptionFailure(Failure failure) — вызывается, если не выполнено условие в методе класса **Assume**;

testRunFinished(Result result) — вызывается после завершения всех тестов. Параметр **result** содержит информацию о числе успешных/провальных тестов, о времени выполнения и проч.;

testRunStarted(Description description) — вызывается перед запуском всех тестов.

Примерная реализация класса обработчика событий:

```
/* # 15 # реализация обработки событий # ScholarshipRunListener.java */
package test.by.bsu.calculation;
import org.junit.runner.Description;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
import org.junit.runner.notification.RunListener;
public class ScholarshipRunListener extends RunListener {
    @Override
    public void testStarted(Description description) throws Exception {
        System.out.println("тест стартовал: " + description.getMethodName());
    }
    @Override
    public void testFinished(Description description) throws Exception {
        System.out.println("тест завершен: " + description.getMethodName()
            + "\n---");
    }
    @Override
    public void testFailure(Failure failure) throws Exception {
        System.out.println("тест провален с исключением: "
            + failure.getException());
    }
    @Override
    public void testIgnored(Description description) throws Exception {
        System.out.println("тест игнорирован: " + description.getMethodName()
            + "\n---");
    }
    @Override
    public void testRunFinished(Result result) throws Exception {
        System.out.println("результаты выполнения тестов:");
        System.out.println("время выполнения: (" + result.getRuntime()
            + ") millis");
        System.out.println("было запущено тестов: " + result.getRunCount());
        System.out.println("провалено тестов: " + result.getFailureCount());
        System.out.println("игнорировано тестов: " + result.getIgnoreCount());
        System.out.println("все тесты завершены успешно: "
            + result.wasSuccessful());
    }
}
```

Регистрация обработчика событий:

```

/* # 16 # регистрация обработчика # ScholarshipRunner.java */

package test.by.bsu.calculation;
import org.junit.runners.BlockJUnit4ClassRunner;
import org.junit.runners.model.InitializationError;
import org.junit.runner.notification.RunNotifier;
public class ScholarshipRunner extends BlockJUnit4ClassRunner {
    private ScholarshipRunListener runListener;
    public ScholarshipRunner(Class<ScholarshipRunListener> clazz)
        throws InitializationError {
        super(clazz);
        runListener = new ScholarshipRunListener();
    }
    public void run(RunNotifier notifier) {
        notifier.addListener(runListener);
        super.run(notifier);
    }
}

```

Для демонстрации документирования обработки событий использован класс с тестами **ScholarshipCalculatorTestRule**, несколько измененный так, чтобы были задействованы все predefined методы класса **RunListener**, а именно: успешный тест, заведомо неудачный тест и пропускаемый тест.

```

/* # 17 # тест, запускаемый с обработкой событий # ScholarshipCalculatorTestRule.java */

package test.by.bsu.calculation;
import java.io.File;
import java.io.IOException;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import org.junit.rules.TemporaryFolder;
import org.junit.rules.Timeout;
import org.junit.runner.RunWith;
import by.bsu.calculation.ScholarshipCalculatorImpl;
@RunWith(ScholarshipRunner.class)
public class ScholarshipCalculatorTestRule {
    private static ScholarshipCalculatorImpl scholarshipCalculator;
    @BeforeClass
    public static void initScholarshipCalculator() {
        scholarshipCalculator = new ScholarshipCalculatorImpl();
    }
    @Rule
    public final TemporaryFolder folder = new TemporaryFolder();
    @Rule
    public final Timeout timeout = new Timeout(300);
}

```

```

@Rule
public final ExpectedException thrown = ExpectedException.none();
@Test
public void writeResultTest() throws IOException {
    File file = folder.newFile("a:/result.txt"); // тест будет провален
    scholarshipCalculator.writeResult(file);
}
@Test
public void stepUpCoefficientCalculateTest() throws NotSuchMarkException {

    thrown.expect(NotSuchMarkException.class);
    scholarshipCalculator.stepUpCoefficientCalculate(11);
}
@Ignore("this test is not ready yet")
@Test
public void writeResultTestTwo() throws IOException {
    thrown.expect(NullPointerException.class);
    scholarshipCalculator.writeResult(null);
}
}

```

В результате отработки теста обработчиком событий в консоль будет выведен отчет:

тест стартовал: stepUpCoefficientCalculateTest

тест завершен: stepUpCoefficientCalculateTest

тест стартовал: writeResultTest

тест провален с исключением: java.io.IOException: The filename, directory name, or volume label syntax is incorrect

тест завершен: writeResultTest

тест игнорирован: writeResultTestTwo

результаты выполнения тестов:

время выполнения: (57) millis

было запущено тестов: 2

провалено тестов: 1

игнорировано тестов: 1

все тесты завершены успешно: false