17. Тестирование (обзор, тестирование разработчиком)

Информатика

1 курс 1 семестр, ИТИС

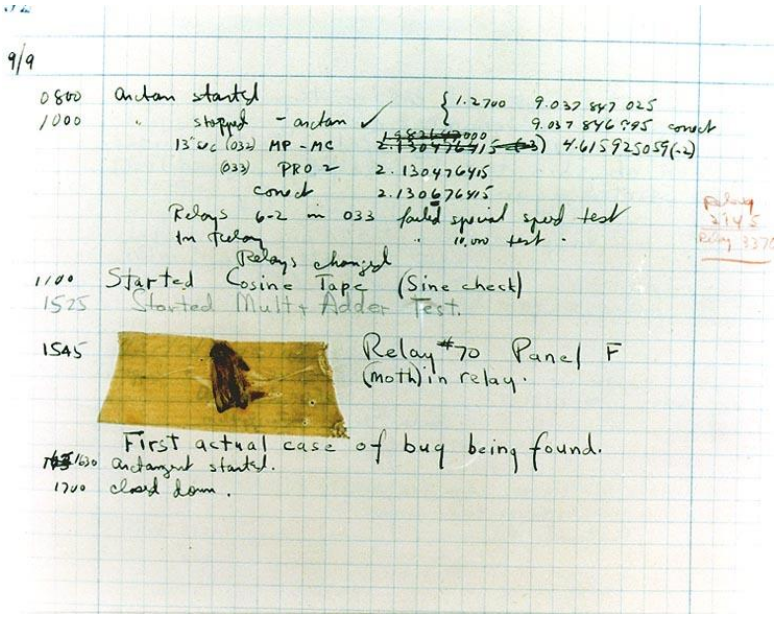
М.М.Абрамский

2017

Bug

- 1878, Томас Эдисон
  – «Неуловимая техническая ошибка»

- 1946, Грейс Хоппер (Harvard)
  – «Первый реальный случай обнаружения бага»
  – Mark II, застрявший в реле мотылек.



Тестирование

- Процесс обнаружения багов

- Лишь часть цикла:
  - Разработка
  - Тестирование
  - Воспроизведение [бага]
  - Исправление/отладка

О каких «тестированиях» мы слышали?

1. Функциональное тестирование
2. Тестирование безопасности
3. Тестирование взаимодействия
4. Тестирование совместимости
5. Нагрузочное тестирование
6. Дымовое тестирование
7. Тестирование сборки
8. Санитарное тестирование
9. Регрессионное тестирование
10. Тестирование установки
11. Тестирование удобства использования
12. Тестирование на отказ и восстановление
13. Конфигурационное тестирование
14. Модульное тестирование
15. Интеграционное тестирование
16. Системное тестирование
17. Приемочное тестирование
18. Альфа-тестирование
19. Бета-тестирование
20. Компонентное тестирование
21. Mock-тестирование
22. Поведенческое тестирование
23. Автоматическое тестирование
24. Ручное тестирование
25. …

Классифицируем

Функциональные
- Функциональное тестирование
- Тестирование безопасности
- Тестирование совместимости

Нефункциональные
- Тестирование установки
- Тестирование удобства использования
- Тестирование производительности
- Тестирование на отказ и восстановление
- Конфигурационное тестирование

Связанные с изменением
- Дымовое тестирование
- Санитарное тестирование
- Регрессионное тестирование

Уровни
- Модульное (компонентное) тестирование
  - Mock-тестирование
- Интеграционное тестирование
  - Поведенческое (сценарное) тестирование
- Системное тестирование
- Приемочное тестирование

Этапы разработки
- Альфа-тестирование
- Бета-тестирование

По типу автоматизации
- Автоматическое тестирование
- Ручное тестирование

Разрушаем мифы о тестировании

Сразу два:

- ~~Все тесты пишутся тестировщиками.~~
- ~~Тестировщик работает с кодом.~~

Разрушаем мифы о тестировании

~~Все тесты пишутся тестировщиками.~~

- Есть **тесты**, которые разрабатывает и запускает **разработчик**

~~Тестировщик работает с кодом.~~

- **Тестировщики могут** писать автоматизированные тесты (программировать), могут использовать sql-запросы для проверки данных,
- **Но** практически всегда тестировщик работает с **готовым продуктом** – интерфейсом, приложением, изделием, установочным пакетом – с тем, с чем будет работать и **конечный пользователь.**

В СЕГОДНЯШНЕЙ ЛЕКЦИИ МЫ ЗАНИМАЕМСЯ ТЕСТИРОВАНИЕМ СО СТОРОНЫ РАЗРАБОТЧИКА

Модульное (unit) тестирование

Процесс проверки корректности работы отдельных частей исходного кода (чаще всего методов) программы путем запуска тестов в искусственной среде.

ОСУЩЕСТВЛЯЕТСЯ РАЗРАБОТЧИКОМ!

Test Case

Артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Под тест кейсом понимается структура вида:

Action > **Expected Result** > **Test Result**

Пример:

Open page "login" > Login page is opened > Passed

Источник: protesting.ru

Тест (в случае модульного тестирования)

- Реализация Test Case.
- Код (void метод), который.

1. Воспроизводит некоторые данные / делает предварительные действия.
2. Выполняет тестируемый метод, правильный результат работы которого очевиден автору теста.
3. Выполняет сопоставление полученного результата с ожидаемым (assert).
   - Если «ожидание» и «реальность» совпадают, тест пройден.
   - Если нет – тест завален (чаще всего генерируется специальное исключение - AssertionError).

Оболочки модульного тестирования

- Средства для разработки тестов, включающие:
  - построение,
  - выполнение тестов
  - создание отчетов.

- 1999 - SUnit для Smalltalk (Кент Бек)
- 2002 - JUnit для Java (Эрик Гамма)

Зачем нужно модульное тестирование?

Зачем нужно модульное тестирование?

- Ошибки выявляются в процессе проектирования метода или класса (если TDD);
- Разработчик создает методы и классы для конкретных целей;
- Снижается число новых ошибок при добавлении новой функциональности;
- Тест отражает элементы технического задания, (некорректное завершение теста сообщает о нарушении технических требований заказчика;
- ...

Реализации

- ...
- JUnit 3
- JUnit 4
- JUnit 5

JUnit3

- Обязательно наследование от TestCase.
- Все тестирующие методы начинаются со слова test.
- Вспомогательные методы, обслуживающие тесты (setUp, tearDown), переопределяются от TestCase.

JUnit4

- Наследование от TestCase не нужно.
- Все на аннотациях:
  - Тесты помечаются @Test, называются «*как угодно**»
  - Вспомогательные методы помечаются аннотациями @Before, @BeforeClass, @After, @AfterClass.

- Как называть тесты?
- ~~Как угодно~~ По naming conventions.

- Суффикс "Test" к тестовому классу.
- Тестовые методы желательно должны содержать "should" в названии (по названию понятно, что такое правильный тест):
  – “sumShouldBePositive”
- *As a general rule, a test name should explain what the test does so that it can be avoided to read the actual implementation.*

(Lars Vogel – vogella.com)

Аннотация @Test

- Объявляет метод (обязательно public void) тестовым.
- Аннотация @Test может использовать параметры:
  - **expected** — код в тесте проверяется на генерацию определенного исключения;
  - **timeout** — код в тесте должен работать не более указанного времени (иначе тест завален);

org.junit.Assert

- Проверка «ожидание/реальность» (expected/actual)

- Методы
  - assertTrue
  - assertFalse
  - assertEquals
  - assertArrayEquals
  - assertNotEquals
  - assertSame
  - assertNotSame

  - fail – гарантированное падение теста.

Example. Vector2D

```
public class Vector2D {
    private double x = 0, y = 0;

    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getX() { return x; }
    public void setX(double x) { this.x = x; }

    public double length(){
        return Math.sqrt(x * x + y * y);
    }
}
```

Example. Vector2DTest

```
import org.junit.*;

public class Vector2DTest {

    @Test
    public void newVectorShouldHaveZeroLength() {

        Vector2D v1 = new Vector2D();

        // Проверка ожидания и реальности
        // для double – с точностью
        Assert.assertEquals(0, v1.length(), 1e-9);

    }
}
```

Example. Vector2DTest (JUnit3 version)

```
import org.junit.*;

public class Vector2DTest extends TestCase {

    public void testThatNewVectorShouldHaveZeroLength() {
        Vector2D v1 = new Vector2D();

        // Проверка ожидания и реальности
        // для double – с точностью
        Assert.assertEquals(0, v1.length());

    }
}
```

Runner («запускалка»)

- Процесс запуска тестов по умолчанию не выполняется с запуском функционала – должен быть кем-то запущен отдельно.

- Можно разработчиком вручную:

```
Result result =
     JUnitCore.runClasses(Vector2DTest.class);

for (Failure f : result.getFailures()) {

    System.out.println(f.toString());

}
```

Результат в консоли

- *Пусто, т.к. все успешно*

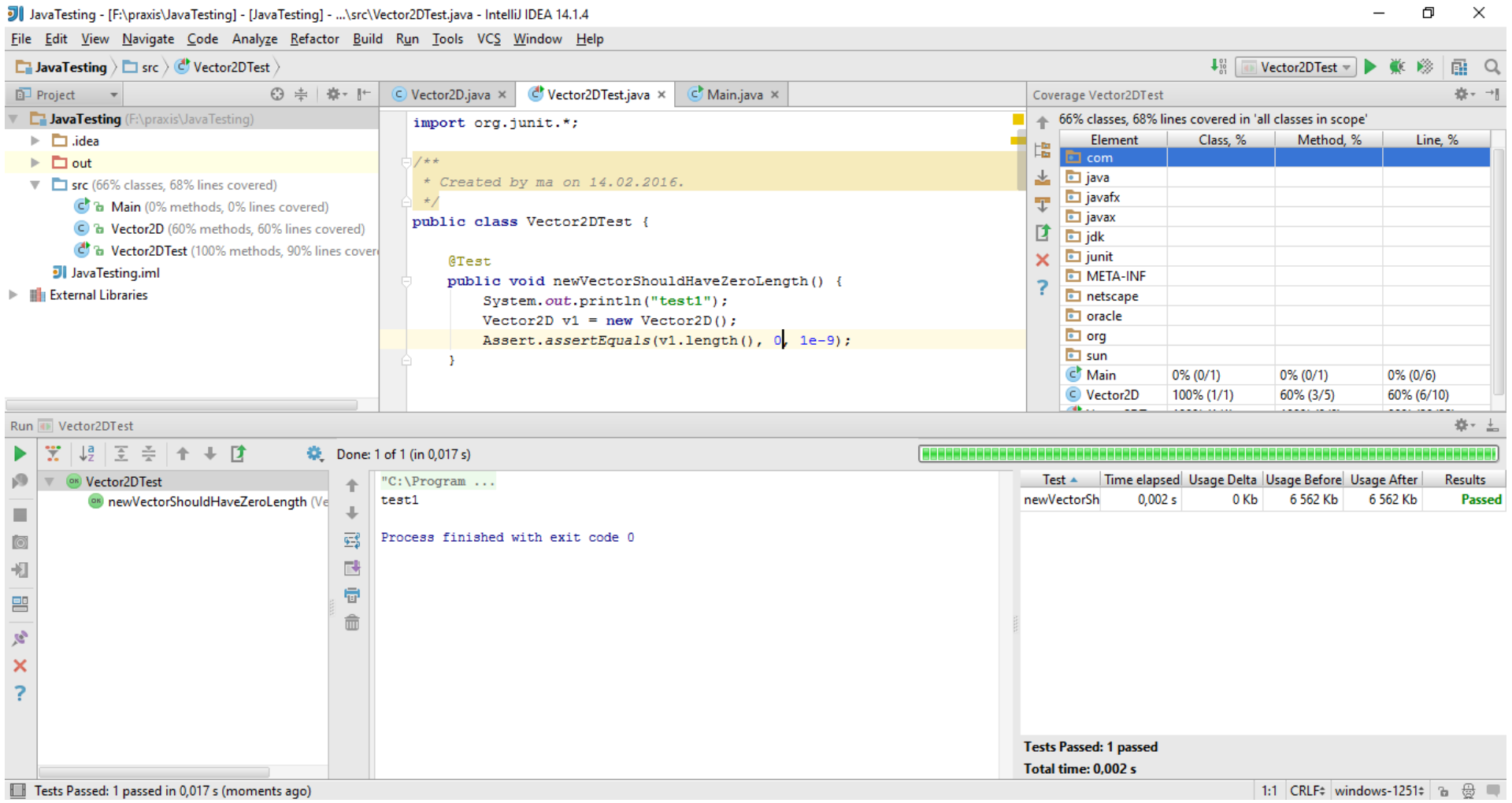
- Если была бы ошибка, вывелся текст:

```
newVectorShouldHaveZeroLength(Vector2DTest): expected:<0.0> but was:<1.0>
```

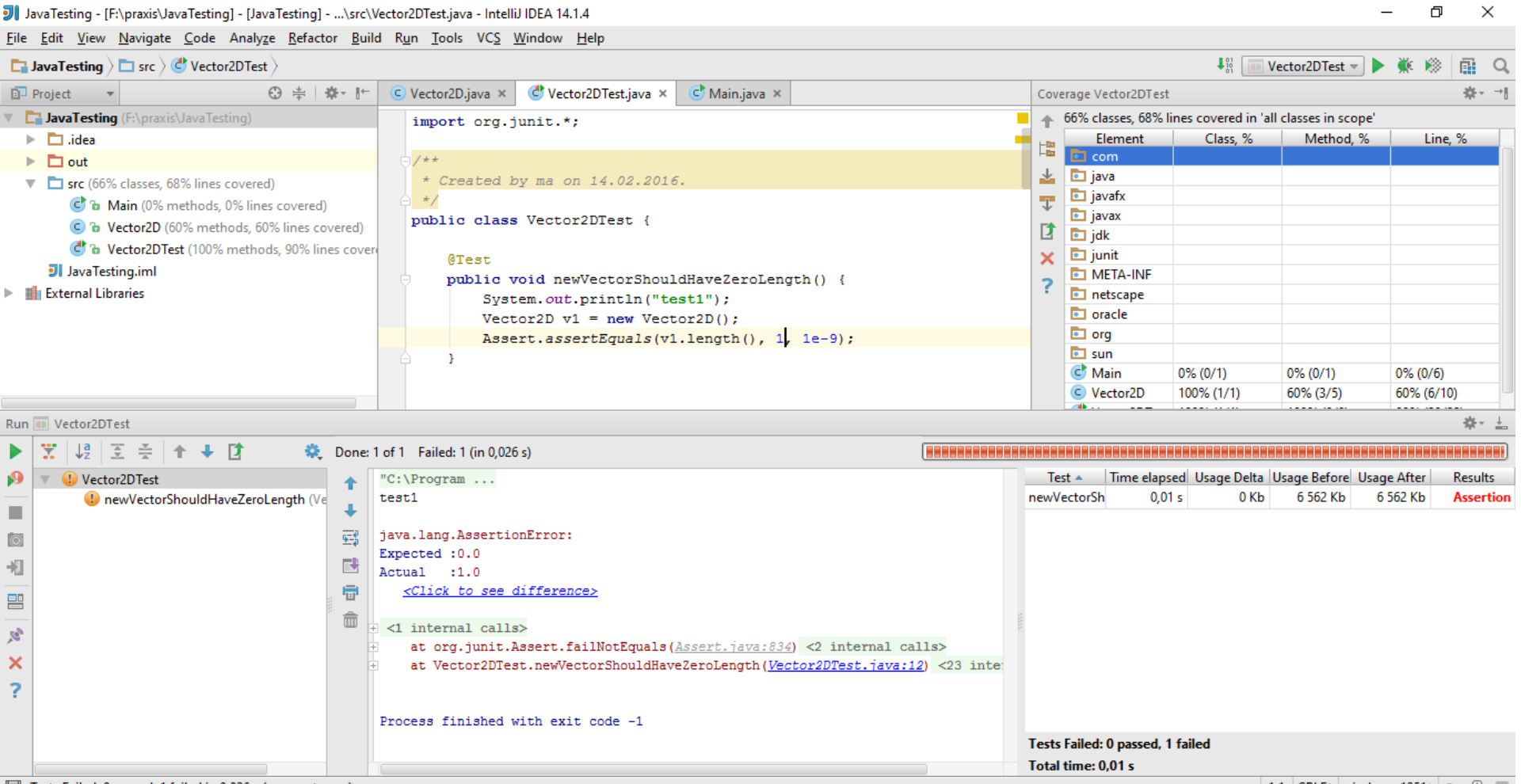
А можно в IDE

- Intellij Idea – опция «Run with coverage» в меню «Run»
- ???
- PROFIT!!!

И - Информативность



И - Информативность



Покрытие (Coverage)

- Процент кода (строк, методов, классов), покрытого тестами.
- А также сами эти тесты.
- *Связано с понятием «покрытия» в дискретной математике и других областях*

Ignore и Assume

- Тест, помеченный @Ignore, не выполняется
  - Злоупотреблять этим не надо!
- Вместо Assert можно использовать Assume (с методами assumeFalse, assumeEquals и др.):
  - Если проверка верна – тест пройден.
  - Если проверка неверна – тест игнорируется.

Фикстура (Fixture)

- *Окружение*, необходимое для корректной работы теста.
  - Объекты, БД, файлы, connect-ы и т.д.

- JUnit4:
  - @BeforeClass — запускается только один раз при запуске теста (static).
  - @Before (setUp в JUnit3) — запускается перед каждым тестовым методом.
  - @After (tearDown В JUnit3) — запускается после каждого метода.
  - @AfterClass — запускается после того, как отработали все тестовые методы (static).

Жизненный цикл тестирующего класса

- @BeforeClass
- Для каждого @Test-метода:
  - создание экземпляра тестового класса
  - @Before
  - @Test
  - @After
- @AfterClass

Дан код. Упростить.

```
public class Vector2DTest {

    @Test
    public void newVectorShouldHaveZeroLength() {
        Vector2D v1 = new Vector2D();
        Assert.assertEquals(v1.length(), 0, 1e-9);
    }

    @Test
    public void simpleVectorObjectShouldHaveZeroX() {
        Vector2D v1 = new Vector2D();
        Assert.assertEquals(v1.getX(), 0, 1e-9);
    }

    @Test
    public void simpleVectorObjectShouldHaveZeroY() {
        Vector2D v1 = new Vector2D();
        Assert.assertEquals(v1.getY(), 0, 1e-9);
    }
}
```

```java
public class Vector2DTest {
    private Vector2D v1;
    private final double EPS = 1e-9;

    // Будет запускаться каждый раз для каждого @Test
    @Before
    public void createSimpleVector() {
        v1 = new Vector2D();
    }

    @Test
    public void newVectorShouldHaveZeroLength() {
        Assert.assertEquals(v1.length(), 0, EPS);
    }
    @Test
    public void simpleVectorObjectShouldHaveZeroX() {
        Assert.assertEquals(v1.getX(), 0, EPS);
    }
    @Test
    public void simpleVectorObjectShouldHaveZeroY() {
        Assert.assertEquals(v1.getY(), 0, EPS);
    }
}
```

```java
public class Vector2DTest {
    private static Vector2D v1;
    private final double EPS = 1e-9;

    // Можно и static
    @BeforeClass
    public static void createSimpleVector() {
        v1 = new Vector2D();
    }

    @Test
    public void newVectorShouldHaveZeroLength() {
        Assert.assertEquals(v1.length(), 0, EPS);
    }
    @Test
    public void simpleVectorObjectShouldHaveZeroX() {
        Assert.assertEquals(v1.getX(), 0, EPS);
    }
    @Test
    public void simpleVectorObjectShouldHaveZeroY() {
        Assert.assertEquals(v1.getY(), 0, EPS);
    }
}
```

```java
public class Vector2DTest {
    private static Vector2D v1;
    private final double EPS = 1e-9;

    @BeforeClass
    public static void createSimpleVector() {
        v1 = new Vector2D();
    }

    @Test
    public void newVectorShouldHaveZeroLength() {
        Assert.assertEquals(v1.length(), 0, EPS);
    }

    // Каждый разработчик хотя бы 1 раз так делал
    @Test
    public void simpleVectorObjectShouldHaveZeroXandY() {
        Assert.assertEquals(v1.getX(), 0, EPS);
        Assert.assertEquals(v1.getY(), 0, EPS);
    }
}
```

Multiple Asserts per One Test

- Возможны, но нежелательны!
  - В случае fail-а одного другие не запустятся
  - Только, если логика кристально прозрачна!
    - Редкое явление.

Expected, Timeout

```
@Test(expected = ArithmeticException.class)
public void checkZeroDenom() {
    // проверяем исключение при делении на ноль
    MyMath.divide(1, 0);
}
```

…

```
// останавливаем тест, если он работает дольше 100 миллисекунд
@Test(timeout = 100)
public void waitMe(){
    while(true);
}
```

Expected, Timeout, Ignore

```
@Test(expected = ArithmeticException.class)
public void checkZeroDenom() {
    // проверяем исключение при делении на ноль
    MyMath.divide(1, 0);
}


// тест игнорируется (вообще не запускается)
@Ignore
// останавливаем тест, если он работает дольше 100 миллисекунд
@Test(timeout = 100)
public void waitMe(){
    while(true);
}
```

Другие возможности

- Группировка тестов – Categories, Suite.
- Запуск параметризованных тестов (для теста заготавливаются наборы данных – Parameterized, Theory.
- Правила – Rules.

Parametrized. Пример. Fibonacci

```
public class Fibonacci {
    public static int compute(int n) {
        int result = 0;

        if (n <= 1) {
            result = n;
        } else {
            result = compute(n - 1) + compute(n - 2);
        }

        return result;
    }
}
```

Пример с https://github.com/junit-team/junit4/wiki/parameterized-tests

Parametrized. Пример. FibonacciTest

```
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {1, 1}, {2, 1}, {3, 2}, {4, 3}, {5, 5}, {6, 8}
        });
    }
    private int fInput, fExpected;
    public FibonacciTest(int input, int expected) {
        fInput = input;
        fExpected = expected;
    }
    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

Пример с https://github.com/junit-team/junit4/wiki/parameterized-tests

Test Driven Development (TDD)

1. Пишем простейший тест, ломающий программу.
2. Пишем простейшую реализацию, достаточную для прохождения теста.
3. *Улучшаем написанный код, не ломая тесты*. Возвращаемся к пункту 1.

Keywords

- Баг, тест, тестирование, fixture, test case, unit testing

Links

http://www.protesting.ru/

JUnit:

- http://www.vogella.com/tutorials/JUnit/article.html (eng)
- http://www.cavdar.net/2008/07/21/junit-4-in-60-seconds/ (eng)
- http://devcolibri.com/864 (rus)
- https://habrahabr.ru/post/120101/ (rus)