Тестирование программных алгоритмов на примере фреймворка JUnit

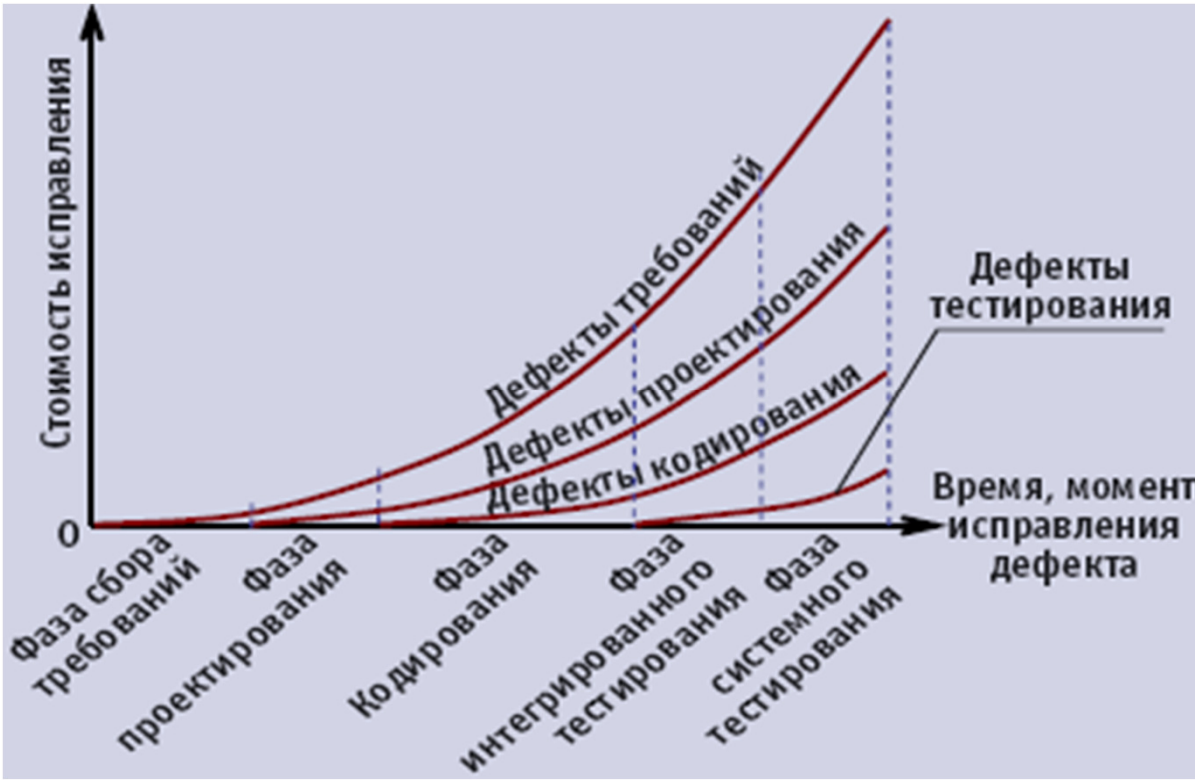
Морозов Николай Сергеевич

Руководитель отдела тестирования и автоматизации «Prolev Technologies»

Проблема:

- Написанный код должен кто – то проверять
- Проверять надо постоянно и быстро
- Входные\выходные\промежуточные данные постоянно забываются
- Кто – то еще хочет безопасно изменить алгоритм или переиспользовать его
- Как и где хранить проверочные значения и тесты

Почему важно постоянно тестировать ПО



Пути решения

Автоматизировать процесс проверки:

- Статистический анализ кода
- Модульные тесты

Пример алгоритма

```
public class Calculator {

   public double add(double a, double b){
      return a + b;
   }

   public double multiply(double a, double b){
      return a*b;
   }
}
```

….

```
Calculator calculator = new Calculator();
calculator.add(2,2);                    // 4.0
calculator.multiply(3, 3);              // 9.0
```

Как придумать тесты?

- Формализованные требования (ТЗ)
- Отраслевые нормы\стандарты\ГОСТ
- Граничные значения
- Позитивные\негативные данные
- Частные решения

Как автоматизировать тесты?

Фреймворк(framework) модульного тестирования Junit:

http://junit.org/

Пример тестов

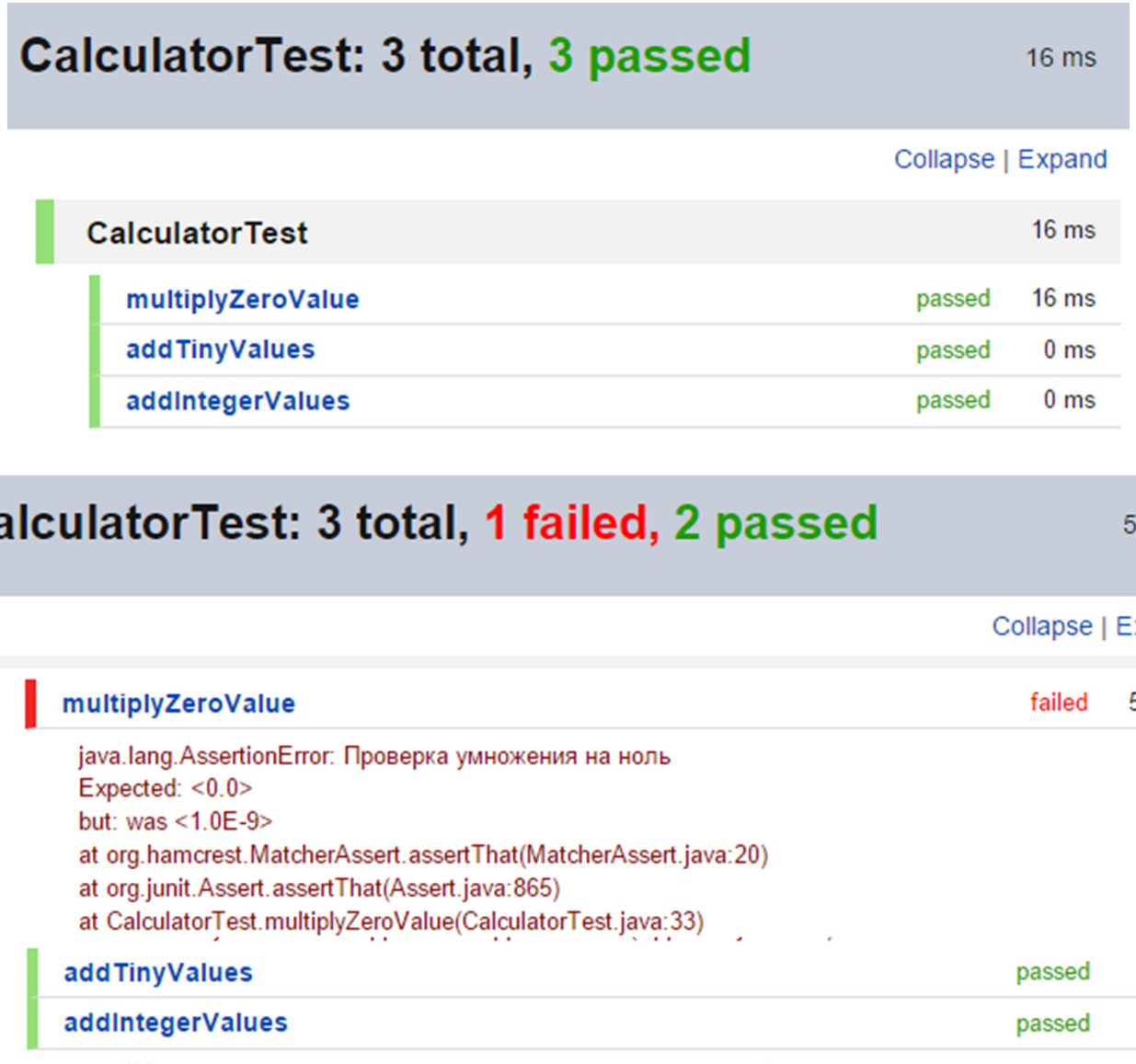
```
public class CalculatorTest {

   @Test
   public void addIntegerValues(){

      Calculator calculator = new Calculator();
      assertThat("Проверка целочисленного сложения", calculator.add(2, 2), equalTo(4));
   }

   @Test
   public void addTinyValues(){

      Calculator calculator = new Calculator();
      assertThat("Проверка сложения малых чисел", calculator.add(0.000001,0.000002), equalTo(0.000003));
   }


   @Test
   public void multiplyZeroValue(){

      Calculator calculator = new Calculator();
      assertThat("Проверка умножения на ноль", calculator.multiply(0,10), equalTo(0.0));
   }
}
```

Запуск

- Из среды разработки (IntelliJ, Netbeans)
- Из систем сборки проектов: Ant, Maven
- Из командной строки:

```
C:\> java org.junit.runner.JUnitCore <test class name>
```

Пример отчетов



Дополнительные возможности

Предварительные действия

```
public class CalculatorTest {

 // Пример
   @BeforeTest
    public void connectToDb(){
    //              …
   }

   @Test
   public void addIntegerValues(){

      Calculator calculator = new Calculator();
      assertThat("Проверка целочисленного сложения", calculator.add(2.0, 2.0), equalTo(4.0));
   }

    @AfterTest
    public void cleanUp(){
   //              …
    }
 // Пример (окончание)
```

Аннотации: @Before, @After, @Ignore

Дополнительные возможности

Матчеры

http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matcher.html

assertThat("**Проверка целочисленного сложения**", calculator.add(2.0, 2.0), **equalTo**(4.0));

assertThat(" **вхождение подстроки**", expectedString, **containsString**(("**abc**"));

assertThat(" **вхождение элемента “,** new String[] {"foo", "bar"}, **hasItemInArray**(**startsWith**("**ba**")))

Преимущества

- Простота использования
- Могут запускать все участники проекта
- Отчеты в разных форматах
- Тесты хранятся отдельно от кода
- Быстрота выполнения
- Документирование функционала

Разработка через тестирование

Разработка через тестирование (англ.**test**-driven development, TDD) — техника **разработки** программного обеспечения, которая основывается на повторении очень коротких циклов **разработки**: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг

Применение в учебном процессе

- Проверка лабораторных работ
- Проверка олимпиадных работ по информатике
- Подготовка дипломных и курсовых работ