IT Systems Security laboratory

Unit tests - JUnit.

Unit tests are used to test the smallest units of code - methods and classes. We write them to check if the developed code is free of errors. A well-developed test shows high efficiency in detecting errors. However, you should be aware that the fact that the test did not find errors does not mean that the code is free of errors. They just were not found. The better the test is developed, the more errors it can detect. However, there are no perfect tests.

Passing the laboratory:
To pass the laboratory, complete the tasks described below. A ready-made, working solution should be presented to the teacher for assessment during classes. Next, the NetBeans project should be compressed and placed in one file, then sent to the faculty e-learning platform within the required deadline. The place where the file should be placed will be indicated by the teacher.

Example

1. Write a program that will determine all prime numbers smaller than 100 according to the sieve of Eratosthenes algoruthm (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) .

To find all prime numbers in a given number range, you can use an algorithm called the Eratosthenes sieve: if the natural number N greater than 1 is not divisible by any prime number not greater than the root of N , then N is a prime number.

To solve the problem lets create two classess. One containing methods doing common operations. Second class - solving the issue.

```
package primenumbers;

public class CommonOperations {

    //calculate the biggest int lower than root value
    int floorRootValue(int number) {
        return (int) Math.sqrt(number);
    }

    //is the number divisable by the other one?
    boolean isDivisible(int divident, int divisor) {
        double result = divident / (double) divisor;
        if (result == (int) result) {
            return true;
        } else {
            return false;
        }
    }
}
```

```
package primenumbers;

import java.util.Scanner;

public class PrimeNumbers {

    CommonOperations co;

    public PrimeNumbers() {
        co = new CommonOperations();
    }

    public int[][] createTable(int x) {
        //initialize and fill the table as follows:
        // |2|3|4|5|6...
        // |1|1|1|1|1...
        int[][] numbersTable = new int[x][2];
        for (int i = 0; i < x; i++) {
            numbersTable[i][0] = i + 2;
            numbersTable[i][1] = 1;
            //value one means that the value in the
            //above row is a prime number
        }
        return numbersTable;
    }

     public int[][] markNonPrimeFromTable(int[][] numbersTable,
int range) {
          //marks non-prime numbers by insterting zeros in the
second row of the table
        // |2|3|4|5|6...
        // |1|1|0|1|0...
        double tmp = 0;
        for (int divisor = 2; divisor <= range; divisor++) {
             for (int i = divisor - 1; i < numbersTable.length;
i++) {
                        if (co.isDivisible(numbersTable[i][0],
divisor)) {
                       numbersTable[i][1] = 0; //mark all non-
prime numbers
                }
            }
        }
        return numbersTable;
    }

    void showPrimeNumbers(int[][] numbersTable) {
        for (int i = 0; i < numbersTable.length; i++) {
            if (numbersTable[i][1] == 1) {
                System.out.print("|" + numbersTable[i][0]);
            }
        }
    }
```

```
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
            System.out.println("Give the maximum of the range
(eg.100)");
        String s = in.nextLine();
        int size = Integer.parseInt(s) - 1;
        PrimeNumbers pm = new PrimeNumbers();
        int[][] numbersTable = pm.createTable(size);
        int root = new CommonOperations().floorRootValue(size);
          numbersTable = pm.markNonPrimeFromTable(numbersTable,
root);
        System.out.println("");
        pm.showPrimeNumbers(numbersTable);
    }

}
```

Now create unit tests for the class. Just generate tests for the CommonOperations class. In NetBeans 8.2 right click the class name, choose tools→ create/update tests (Fig. 1)

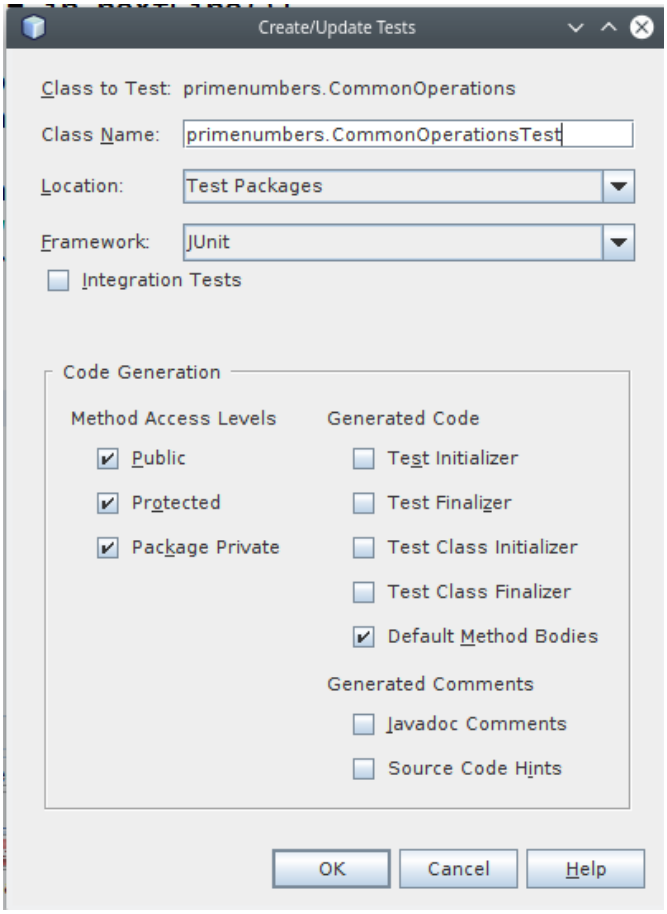

Figure 1. Test class generation

You should see the code as follows:

```
package primenumbers;

import org.junit.Test;
import static org.junit.Assert.*;

public class CommonOperationsTest {

    public CommonOperationsTest() {
    }

    @Test
    public void testFloorRootValue() {
        System.out.println("floorRootValue");
        int number = 0;
        CommonOperations instance = new CommonOperations();
        int expResult = 0;
        int result = instance.floorRootValue(number);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    @Test
    public void testIsDivisible() {
        System.out.println("isDivisible");
        int divident = 0;
        int divisor = 0;
        CommonOperations instance = new CommonOperations();
        boolean expResult = false;
              boolean  result  =  instance.isDivisible(divident,
divisor);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

}
```

If you run these tests (Alt+F6) they will fail (Fig. 2). It is necessary to correct the test methods.

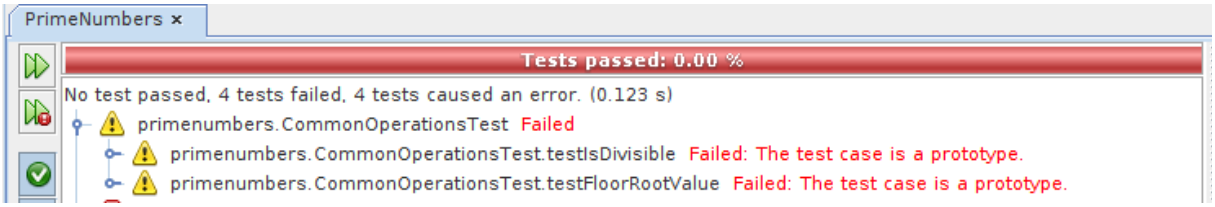

Figure 2. Test execution result

We will start from the method testIsDivisible(). To make it working it is necessary to give correct values as parameters and define expected result(Listing 1). Afer running the tests once again we will see the test result OK (Figure 3).

Listing 1. testIsDivisible method body – version 1.

```
@Test
public void testIsDivisible() {
    System.out.println("isDivisible");
    int divident = 4;
    int divisor = 3;
    CommonOperations instance = new CommonOperations();
    boolean expResult = false;
    boolean result = instance.isDivisible(divident, divisor);
    assertEquals(expResult, result);
}
```

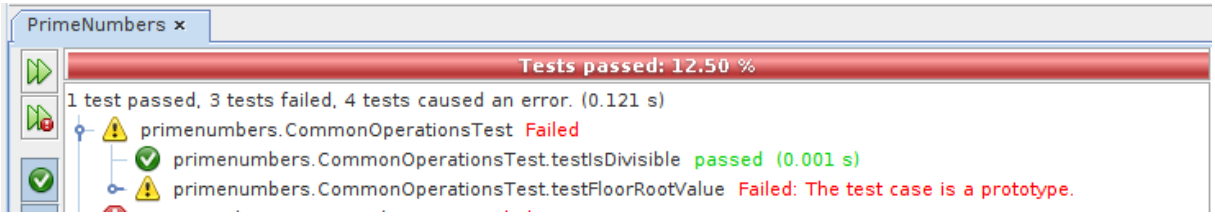

Figure 3. Test execution result

Now lets correct the second test (Listing 2).

Listing 2. testFloorRootValue method body – version 1.

```
@Test
public void testFloorRootValue() {
    System.out.println("floorRootValue");
    int number = 5;
    CommonOperations instance = new CommonOperations();
    int expResult = 2;
    int result = instance.floorRootValue(number);
    assertEquals(expResult, result);
}
```

As we can see the corrected tests pass. Unfortunately it is not enough. It is necessary to run the tests giving various parameters values to the methods and checking various results (see the lecture).
It is time to parametrize tests. To mke it possible it is necessary to add appropriate library to test libraries. We will use JunitParams library (Figure 4).

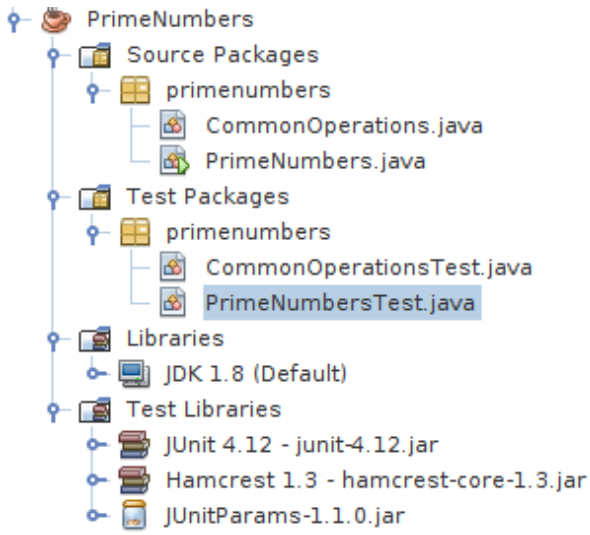

Figure 4. Project tree with JunitParams library added.

Now it is possible to define that all tests will be run with JunitParams library usage. To do so it is enough to place appropriate annotation before the class (Listing 3). Please do not forget to add necessary imports.

Listing 3. Using runner

```
import junitparams.JUnitParamsRunner;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;

@RunWith(JUnitParamsRunner.class)
public class CommonOperationsTest {
```

Now it is possible to parametrize test method. To do so lets put @Parameters annotation before the method. Inside annotation's brackets you can give parameters for the test method. It implies that the test method has to accept given parameters and use their values apropriately. The example of parametrized method is presented in listing 4.

Listing 4. Parametrized test method

```
@Test
@Parameters ({"5,2","9,3"})
public void testFloorRootValue(int number, int expResult) {
    System.out.println("floorRootValue");
    CommonOperations instance = new CommonOperations();
    int result = instance.floorRootValue(number);
    assertEquals(expResult, result);
}
```

Naming Conventions

Till now we were using the default names of thest methods. According to conventions we should use descriptive names that explain what particular tests are doing. More in this subject on https://dzone.com/articles/7-popular-unit-test-naming.
We will focus on one standard. Please see the fragment of mentioned webpage:
„MethodName_ExpectedBehavior_StateUnderTest: Slightly tweeked from above, but a section of developers also recommend using this naming technique. This technique also has disadvantage that if method names get changed, it becomes difficult to comprehend at a later stage. Following is how tests in first example would read like if named using this technique:

- isAdult_False_AgeLessThan18
- withdrawMoney_ThrowsException_IfAccountIsInvalid
- admitStudent_FailToAdmit_IfMandatoryFieldsAreMissing".

Lets change names of our test methods according to this standard. The names can be as presented in listing 5.

Listing 5. Example names of test methods

```
@Test
@Parameters ({"5,2","9,3"})
public void floorRootValue_returnsCorrectResult_correctParameterGiven(int numl

@Test
public void isDivisible_True_isDivisible() {...9 lines }

@Test
public void isDivisible_False_isNotDivisible() {...9 lines }
```

Equivalence class, boundary conditions and exceptions

Although the tests we have writen prove that methods work well, they don't check methods

behaviour in all situations. Eg. We do not test what will happen if a zero value is given as a divisor to the method isDivisible(). Now we should add all necessary tests to fully test all methods from the class. To check if the method isDivisible() throws exception when we try division by zero we need an additional test. In the test we can add a parameter to the @Test annotation (Listing 6). After running the test we will find that the method doesn’t behave properly(Figure 5).

Listing 6. Testing if the method throws exception

```
@Test(expected=IllegalArgumentException.class)
public void isDivisible_throwsIllegalArgumentException_whenDividedByZero() {
    System.out.println("isDivisible");
    int divident = 4;
    int divisor = 0;
    CommonOperations instance = new CommonOperations();
    boolean result = instance.isDivisible(divident, divisor);
}
```

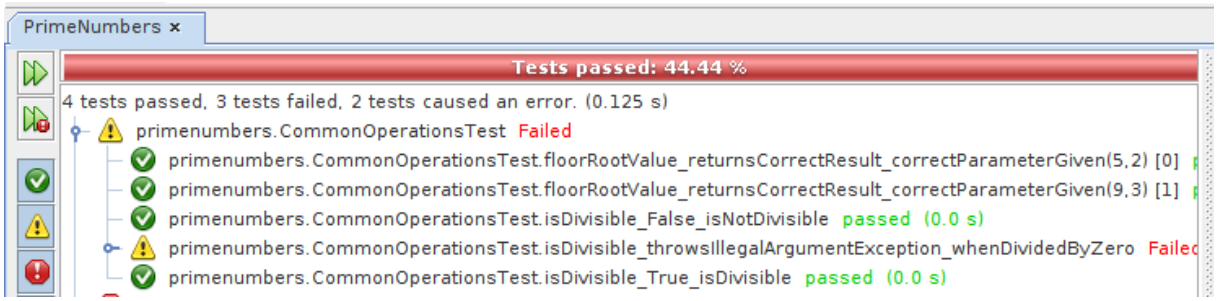

Figure 5. Test result

In order to obtain proper behaviour of the method it is necessary to introduce some changes in the method (Listing 7).

Listing 7. Method isDivisible() with throw exception instruction

```
boolean isDivisible(int divident, int divisor) {
    if (divisor==0)
        throw new IllegalArgumentException();
    double result = divident / (double) divisor;
    if (result == (int) result) {
        return true;
    } else {
        return false;
    }
```

To completly test the method we have to take under consideration equivalence partitions and boundary values. Please take under consideration a code coverage with tests (see the lecture).

Code isolation

Now we want to test the markNonPrimeFromTable method. Unfortunately this method calls isDivisible method. We have to isolate these methods. In this example we will use inline stub. To make it possible to define such a stub, we have to redefine the class PrimeNumbers to make it possible to give to this classs a stub of the isDivisible method. In order to do this we will create an additional constructor of PrimeNumbers class that accepts as a parameter the CommonOperations class object (Listing 8).

Listing 8. PrimeNumbers class constructors

```
public class PrimeNumbers {

    CommonOperations co;

    public PrimeNumbers() {
        co = new CommonOperations();
    }

    public PrimeNumbers(CommonOperations co) {
         this.co = co;
    }
```

Now, in the test we can create a stub of the method isDivisible() and use it in a test method. In the example presented in the listing 9 an inline stub was used.

Listing 8. A test with inline stub

```
@Test
public void testMarkNonPrimeFromTable_MarksNoNPrime_WhenNonDivisible() {
    System.out.println("markNonPrimeFromTable");

    //given
    int[][] numbersTable = {{2, 1}, {3, 1}, {4, 1}, {5, 1}};
    int range = 2;
    int[][] expResult = {{2, 1}, {3, 0}, {4, 0}, {5, 0}};
    PrimeNumbers instance = new PrimeNumbers(new CommonOperations() {
        boolean isDivisible(int divident, int divisor) {
            return true;
        }
    });

    //when
    int[][] result = instance.markNonPrimeFromTable(numbersTable, range);

    //then
    assertArrayEquals(expResult, result);
}
```

Tasks to do:
1.Parametrise all methods that should be parametrized
2. Rename test methods to give them names according to the choosen convention
3. Add all necessary tests to obtain good quality tests. Take under consideration equivalence partitions and boundary values.
4. Add tests to the PrimeNumbers class. Use various techniques of code isolation.
5. Write one more program that counts points and determines rounds in bowling game for two players. The program should behave as a table over the bowling track. Use TDD technique. Pay an attention to use apropriate test cases. Rules of bowling game you can find in the internet – eg. https://slocums.homestead.com/gamescore.html