Workshop 2 : JUnit testing & TDD

In this exercise, we will cover the functionality of the testing framework JUnit and how basic unit tests are written for the framework. We will also briefly evaluate TDD (Test-Driven Development)[1] and discuss the advantages of this paradigm. To truly understand the advantages of TDD, the exercise needs to be done in a group.

The JUnit framework is fully supported by NetBeans and will work without the need of installing a plugin.

1 Preparation

We will work with a predefined class, which we will test, error correct and extend in small iterative steps.

1. Download the NetBeans project "test.zip" from the course home page and import it into NetBeans.

2. Inspect and read the code comments. Take special note of the toString() method inside the Node class. This is a good method to use during debugging and development.

   If you look closely in the project view, you can see that the test code and source code are clearly separated in their own directories.

2 Testing of the List class

Now that the project is prepared and imported into NetBeans, we can begin to write some JUnit tests. The first unit test we create will test the add() method of the List class.

1. Create the class "ListTest" that will hold the unit tests for the List class. You can create this file via a template in NetBeans by right-clicking on the project and selecting; New > Other... > Unit Tests > Test for Existing Class.

   a) Select the List class under "Class to Test". If you look closely; the default behaviour is to place the test class in the same package as the class you selected. Any classes part of the same package as the test class, will be automatically reachable from the tests without the need of an import. Using this structure also gives us the ability to test package-private classes.

   b) Disable the check boxes under "Generated Comments" and the check box "Default Method Bodies".

   c) Press "Finish" to create the test class.

[1] Wikipedia, Test-driven development: http://en.wikipedia.org/wiki/Test-driven_development

2. Inspect the newly generated test class. If you look at the signatures of the generated methods, you can see that there is a testAdd() method. This is the method we want to use when testing the add() method of the List class.

3. Run the unit tests by going to the top menu and executing; Run > Test Project. As the unit tests inside the test class are empty, NetBeans should report that the tests pass.

4. Edit the testAdd() method to look like the following piece of code:

```
@Test
public void testAdd() {
    List l = new List();
    l.add(1);
    assertTrue(l.getLength() == 1); // The logical check
}
```

   The last row in the method is called an assertion. If the assertion holds true, it means that the unit test was successful. If, on the other hand, the assertion fails (and does not hold true), an error is reported.

   NOTE! This is how all unit tests work. The tests either fail or succeed. Manual reporting (using for example System.out.println()) should never be needed[2].

5. Run the tests of the project again. The tests should pass this time as well. If any tests fail, they are shown in the test report at the bottom of the NetBeans window. If you want to test this, you can induce a failure by modifying the assertion inside the testAdd() method to something that should fail.

   NOTE! The testAdd() method is actually testing a void method. This means that there is no return value that can be fetched from the method. Consequently, there needs to be some other way to check if the method operates properly (in this case using getLength()). There isn't always a public method available for this. In those specific cases, a method can be added to the class in order for it to be testable. If that isn't an option, Java reflection can instead be used to access private fields and methods of the class being tested.

6. Next, let's write the test for the remove() method of the List class. The method removes the first node in the List and returns the content (or value) of that node.

   Make use of the return value when you write the test. You should be able to create a test that fails. When you succeed in getting a failure, you should modify the List class accordingly in order to get the test to pass.

7. You should now write a test for the get() method. You should do the following operations in the test:

[2] An exception is during development and under heavy testing when writing the unit test. However, the printout should always be removed once the test is completed.

a) Add five values to the list.

b) Return (and check) the value for index 2 in the list.

Something is wrong with the get() method and it does not work correctly. We assume that we do not know why and use the debugger in order to try to find out what's wrong:

a) Set a break point on the following row inside the get() method of the List class:

```
Node<Integer> pos = head
```

Do this by clicking in the left margin of the editor window. This should display a red square right beside the row. The square shows that there is a break point set for this row.

b) In order for the break point to work, we need to run the tests in debug mode. This can be accomplished by selecting "Debug > Debug Test File" in the main menu of NetBeans.

The execution should stop at the break point you specified. You can now use the debugger in order to figure out where the get() method fails. The top toolbar offers a variety of ways to control the execution during debugging. You can also use the mouse to inspect current values of variables just by hovering over them in the editor window. The bottom (Variables) view also let's you browse instances declared in the current scope.

c) Experiment with the debugger until you find the bug in the get() method. Correct the error and make sure the unit test passes.

8. It is also important that we test that methods behave correctly (and report an error) when they get invalid data. The get() method of the List class actually has such an error check. To test it, we can create the following test:

```
@Test(expected=IllegalArgumentException.class)
public void testGetBadIndex() {
    // Get a list then ...
    list.get(-1); // Exception!!!
}
```

The test checks that the expected exception occurs when we request an index that doesn't exist.

9. Create the method copy() inside the List class. This method should create and return a deep copy of the list.

10. Create a unit test for the copy() method.

3 Fixtures

Many tests need some kind of initialization before they can run. In certain cases, a test method might also need to run some shut down code in order to free up resources that were needed during the test. Operations such as these can be done inside special methods that are executed by JUnit in preparation of each test. These are called fixtures.

1. To try out this feature, add the code below to the ListTest class. Note that the printouts below were added just for demonstration purposes and should never be part of a JUnit test class under normal conditions.

```
@BeforeClass
public static void beforeClass(){  //First of all
    System.out.println("Before class");
}
@AfterClass
public static void afterClass(){   //Last of all
    System.out.println("After class");
}
@Before
public void before(){  //Before each test method
    System.out.println("Before");
}
@After
public void after(){ //After each test method
    System.out.println("After");
}
```

4 A quick introduction of TDD

Test-Driven Development aims to drive the development of the code, instead of the other way around (something which is usually the case). Consequently, tests are created and formulated before any implementation (the actual code being tested) of the class is written. The test then becomes the template that describes how the implementation is intended to function. This is usually done in small iterations, with the developers switching between writing the tests of the class and developing the actual code being tested[3].

The main intention with this workflow is to avoid that important parts of the implementation are overlooked while also minimizing the risk of new bugs during the development of the project.

[3] Wikipedia, Test-driven development cycle: http://en.wikipedia.org/wiki/Test-driven_development#Test-driven_development_cycle

1. A popular method used when learning the basics of TDD is to divide the work between two parties. One party writes the test, while the other party writes the implementation. We will use this workflow in the following exercise. Each assignment below is marked for the party that it is intended for.

2. *[Party 1]* Create the class PrimerTest together with the test method isPrimeTest(). This method is intended to test a method with the signature *boolean isPrime(int number)*.

   a) Write the test and check that isPrime() returns true or false depending on if the supplied value is a prime number.

   b) Create the Primer class (the implementation). You should only create an empty skeleton class with an empty isPrime() method; enough to make the code compile. At this stage, the test should not succeed. Make sure that the test class and implementation class are part of the same package.

3. *[Party 2]* Implement the Primer.isPrime() method and try to make the PrimerTest suceed.

4. *[Party 2]* Write the test method PrimerTest.getPrimeTest(). This method is intended to test a method with the signature *int getPrime(int sequenceIndex)*. Again, we only create an empty method inside the Primer class. The method should later return the prime number *sequenceIndex* within the sequence of prime numbers. The following should hold true:

   a) getPrime(0) == 2

   b) getPrime(9) == 29

   c) getPrime(< 0) should through a IllegalArgumentException

5. *[Party 1]* Implement the Primer.getPrime() method and try to make the PrimerTest succeed. Try to implement the method as trivialy as possible.

   NOTE! A very simple (albeit not very optimal) way to implement the getPrime() method is to simply call the isPrime() method in a loop.

6. If possible, optimize and re-factorize the Primer class. This can now be done without the risk of new bugs being introduced. Run the unit tests continually while modifying the implementation code.

5 Test coverage

Unit tests are intended to be a tool for preventing that new bugs creep into classes when new code is added. They should also work as a description over how the implementation being tested is intended to function. The only way to satisfy these requirements is to strive for full test coverage. Consequently, this means that the implementation needs to be tested and traversed as thoroughly as possible.

Achieving full test coverage, can obviously be extremely difficult, considering all the conditions and nooks that a method or class can contain. Luckily, there are great test coverage tools available for Java that can help us in this regard. Two such tools include Cobertura[4] and JaCoCo[5]. Tools such as these offer invaluable help during the development of your project.

[4]Cobertura: http://cobertura.sourceforge.net/
[5]JaCoCo Java Code Coverage Library: http://www.eclemma.org/jacoco/