Workshop 2 : JUnit testing

In this exercise, we will cover the functionality of the testing framework JUnit and how basic unit tests are written for the framework. The JUnit framework is fully supported by NetBeans and will work without the need of installing a plugin (most IDE's support JUnit, use any you like)

Preparation

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

1. Download the NetBeans project ”test.zip” (a Maven project) from the course home page and import it into NetBeans (or into you IDE of choice).
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.
3. Read the README-file.

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 icon and selecting; New > Other... > Unit Tests > Test for Existing Class.

2. Select the List class under ”Class to Test”. 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.

3. Disable the check boxes under ”Generated Comments” and the check box ”Default Method Bodies”. Press ”Finish” to create the test class.

4. 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.

5. Run the unit tests by right click in code > Test File. As the unit tests inside the test class are empty, NetBeans should report that the tests pass (a window will show up in NetBeans).

6. 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 except during development and under heavy testing when writing the unit test.
   However, the printout should always be removed once the test is completed..

7. Run the tests of the project again. The tests should pass this time as well.

   You can induce a failure by modifying the assertion inside the testAdd() method to something that should fail. Do and run again!

   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 may 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, not covered in course).

8. 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.

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

   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. We will use the debugger in order to try to find out what's wrong.

10. Set a breakpoint (by double click in the right margin) at the following line inside the get() method of the List class:

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

    This should display a red square right beside the line. The square shows that there is a breakpoint set for this row.
    In order for the breakpoint 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 breakpoint 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.
    Experiment with the debugger until you find the bug in the get() method. Correct the error and make sure the unit test passes.

11. 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.

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

    Create a unit test for the copy() method.

Fixtures

Many tests need some kind of initialization before they can run. In certain cases, a test method might also need to run some shutdown 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”);
}
```