Exercise 2: Developer Testing with JUnit

This exercise introduces testing by developers and the JUnit test framework for unit testing.

2.1 RingBufferTest (2 points)

The implementation of a *RingBuffer* (from the draft version of *Introduction to Programming in Java: An Interdisciplinary Approach,* R. Sedgewick and K. Wayne, Addison-Wesley, 2008) is the class to be tested. The code is available on elearning: *RingBuffer.java*.

Write unit tests for the class *RingBuffer*. Use a test fixture (setup) providing a **RingBuffer containing three elements**. Each of the test methods should use this fixture.

Keep following points in mind:

- Name the test class containing the tests *at.fhhagenberg.sqe.exercise2.RingBufferTest*.
- Implement at least 3 different test cases (test methods in the class *RingBufferTest*), that test different aspects of the implementation.
- Note: Name the Eclipse project SQE02-*Lastname*_RingBufferTest; replace *Lastname* by your last name.

2.2 EmptyRingBufferTest (1 point)

Write another test class *EmptyRingBufferTest* containing tests for an empty RingBuffer as test fixture (setup). Again, the fixture should be used in each test method. Please note:

- Name the test class *at.fhhagenberg.sqe.exercise2.EmptyRingBufferTest.*
- Provide at least one test case. This test should provoke an exception. Make sure the expected is actually thrown and check the exception type, message, etc.

2.3 AllTests (1 point)

Create a test suite (class *at.fhhagenberg.sqe.exercise2.AllTests*) to run all the tests implemented as part of this exercise.

The test suite can be generated by Eclipse: Create a new test suite via *File* > *New* > *Other...* > *JUnit* > *Test Suite* and check the available test classes to add them into the suite.

Run the test suite (like any other test class) and make sure that it contains all tests.

Exercise 3: Refactoring & Developer Testing

The goal of this exercise is to show the need and benefit of design and refactoring towards clear interfaces for effective unit testing.

3.1 Extracting a domain class Triangle (2 points)

In exercise 1 the application *Triangle1st* containing a graphical UI has been developed (see *fhhagenberg.sqe.exercise1)*. Refactor the design of your implementation of this application in order to extract a class *Triangle* that subsumes all methods and attributes relevant for working with triangles. Use the refactoring support offered by Eclipse. Furthermore, consider following points:

- Implement a constructor *public Triangle(a, b, c)* for creating new *Triangle* objects by specifying the three sides *a, b* and *c*. The constructor should only instantiate valid triangles. If an invalid parameter combination is specified, a meaningful exception should be thrown, giving the reason as message.
- Avoid that invalid triangles can be created, e.g., by overwriting the default constructor *Triangle()* as private.
- Add a static member *boolean isValid(a, b, c)* allowing the user to check the validity of a given parameter configuration before using the constructor.
- Add the methods for calculating the perimeter and the area as well useful getter methods.
- Provide *Javadoc* comments for all methods of the class *Triangle*.

3.2 Project Structure (1 point)

The refactoring should make the application's model (the class *Triangle*) explicit. So the application is split into layers, making the model independent from the graphical UI.

- Reflect the two layers by following package structure

  fhhagenberg.sqe.exercise3. *ui* ..... for the graphical UI

  model ..... for the model

  and move the classes in the according packages. Use the refactoring of Eclipse: *Right-Click > Refactor > Move ….*
- Rename the project to fit following convention: SQE03-*Lastname*_Triangle1st.
- Create a further Eclipse Project SQE03-*Lastname*_Triangle1stTest for the tests. Use the same package structure as in the application project. Define a dependency from the test project to the application project by *Right-Click – Build Path > Configure Build Path ... > Projects > Add ...*

3.3 Unit Tests for class Triangle (2 points)

Use *JUnit* to create tests for the model, i.e., the class *Triangle*. Make sure that all methods of *Triangle* are tested. Add these tests to the test project.

3.4 Dependency Injection (1 point)

Read the article *J.B. Rainsberger: Injecting Testability into your designs (Better Software, April 2005)*. This article shows the impact of unit testing on the software design. Answer the following questions and document the answers in the text file *Exercise3-Q&A.txt*.

- What symptoms indicated the need for improving the design in the scenario described in the article?
- What are the three possible locations for injecting dependencies mentioned in the article?

Submission Instructions

- Add a comment at the beginning of every source code file containing your full name and your student ID number.
- Submit the complete Eclipse projects including all relevant additional files as specified via packing in a Zip archive. Adhere to the naming convention for projects: SQE##-*Lastname_Title* (## = exercise number, *Lastname* = your name, *Title* = exercise title).
- The Eclipse projects have to be completely self-contained. Do not use absolute paths to reference libraries or otherwise the submitted exercise cannot be compiled without build errors and will not be accepted.
- Upload the archive at the specified submission link on the elearning platform http://elearning.fh-hagenberg.at/ until the specified date, usually Tuesday 23:55 before the next lecture.
- Only complete submissions adhering to all of the above requirements are considered. Late submissions, submissions via email or submissions failing to meet the specified requirements will not be accepted.