Exercise 6: Coverage Analysis and White-box Testing

The objective of this exercise is the application of methods for analyzing the test coverage in order to design white-box test cases.

6.1 Creating White-box Test Cases (3 points)

In the previous example you have developed test cases for the class *RingBuffer*. Extend these test cases by white-box tests in order to **cover all instructions and all branches** of the implementation of the *RingBuffer*.

Use the coverage tool *Emma* (http://emma.sourceforge.net/), which is available as plugin for Eclipse as *EclEmma* (http://www.eclemma.org/) and/or can be installed via *Help > Eclipse Marketplace ... > Find: "EclEmma"*

First, measure the coverage of the test cases you previously wrote for the *RingBuffer* with the help of the tool *EclEmma*. Then, write additional test cases until a full coverage has been achieved for all instructions and branches. Keep in mind:

- ☐ Create a new Eclipse project SQE06-*Lastname*_RingBufferTest; replace *Lastname* by your last name. Copy the class *RingBuffer* and your existing test classes to this project.
- ☐ Open the Eclipse view showing *EclEmma* via *Window > Show View > Other… > Java > Coverage*. Start *EclEmma* via right click on the test class `AllTests`: *Run > Coverage As… > JUnit Test*. Switch to instruction/branches counters in the view menu.
- ☐ Before you implement new test cases, export a coverage report in HTML format for your project. (Right click on the project in EclEmma's coverage view > *Export Session…*)
- ☐ Create a new test class `RingBufferWhiteboxTest` containing further test cases in order to achieve full coverage of the class `RingBuffer`.
- ☐ Extend the test suite `AllTests` to combine the existing tests and `RingBufferWhiteboxTest` for execution.

Submission: Submit (1) the complete Eclipse project including (2) the source code of all test classes and (3) the exported HTML coverage report. Use relative paths for all included jar files and libraries.

6.2 Mutation Analysis (3 points)

Mutation analysis makes small changes to the tested code in order to simulate artificial defects. This changed code is called a "mutant" of the original program. When the mutants are tested with the existing test cases, the number of seeded defects found ("killed mutants") indicates the quality of the tests.

Use the mutation tool *PIT* (http://pitest.org/) to complete the exercise. Install the PIT Eclipse plug-in *Pitclipse* from the update site *http://eclipse.pitest.org/release* via *Help > Install New Software... > Add > Name: "PIT", Location: "http://eclipse.pitest.org/release".*

Use PIT to determine the mutation score for the JUnit test cases you implemented in the previous exercises. Write additional test cases for all undetected mutants. Pay attention to following details:

- ☐ Determine the mutation score for the existing tests by right clicking on the class *AllTests: Run As > PIT Mutation Test*. The generated HTML output is shown in the view *PIT Summary*. This output can be found in the workspace in the directory .metadata/.plugins/org.pitest.pitclipse.core/html_results/... Store a copy of the directory containing the HTML output before adding any further tests.
- ☐ Create a new test class `RingBufferMutationTest` and further test cases to achieve a mutation score of 100%. Add this test class to the test suite `AllTests`. Store a copy of the directory containing the HTML output after adding the necessary tests.
- ☐ Use the Eclipse project SQE06-*Lastname*_RingBufferTest previously created; replace *Lastname* by your last name. Use relative paths for all included jar files and libraries.

Submission: Submit (1) the complete Eclipse project including (2) the source code of *RingBuffer* and the corresponding tests cases. Provide (3) the directories containing the HTML reports before and after adding the test.

6.3 Measuring Coverage (3 point)

Coverage analysis is sometimes used to assess the quality of the tests. But what can coverage tell us about the tests? The (discontinued) open source project *htmlparser* (http://htmlparser.sourceforge.net/) contains about 60 KLOC Java code and more than 600 unit tests. What about them?

Instructions

- ☐ Download HTMLParser-2.0-SNAPSHOT-src.zip (Integration Build 2006-09-23) as provided via the elearning platform (or directly from http://sourceforge.net)
- ☐ Create a new Java project in Eclipse based on the extracted source code from the archive HTMLParser-2.0-SNAPSHOT-src.zip.
- ☐ Proceed as follows to remove building errors: Add JUnit 3 to the build path; exclude *org.htmlparser.util.SimpleNodeIterator.java* located in *lexer/src/main/java* from the build path; exclude *HtmlTaglet.java* located in *src/main/java* from the build path.
- ☐ Run *org.htmlparser.tests.AllTests* to execute the project's tests. If necessary, comment out the methods of repeatedly blocking tests, e.g., *StreamTests.testThreaded()*.
- ☐ Measure the coverage of the tests from the development snapshot of *htmlparser 2.0*. Answer following questions and document your findings (a) by generating a PDF coverage report for each question and (b) by giving a *short* textual explanation in the file *Exercise6-3.txt*.
  1. What is the coverage of the project? Is the coverage of the application code (excluding test code) higher or lower than the coverage of the whole project? Give percentage

values for instruction coverage and explain your answer.

2. Measure the coverage of the unit tests (a) with an established network connection and (b) without network connection. Do the results differ? Give percentage values for instruction coverage and explain potential problems.
3. Measure the type coverage achieved only by the tests in the test class *org.htmlparser.tests.scannersTests.ScriptScannerTest.* How many percent of the project's classes are covered? What are the advantages and disadvantages of covering several different classes in addition to the tested class *ScriptScanner*?

Submission: Submit (1) the coverage reports as well as (2) the document containing your answers *Exercise6-3.txt*.

6.4 Gaining and Sustaining Coverage (1 point)

Consider you are appointed as project manager for a software project. The system developed in this project has already several thousand lines of code but no unit tests. What strategy would you choose to improve test coverage in this project?

Instructions

- ☐ Read the blog entry *Code coverage goal: 80% and no less!* by Alberto Savoia: http://googletesting.blogspot.co.at/2010/07/code-coverage-goal-80-and-no-less.html#!/2010/07/code-coverage-goal-80-and-no-less.html.
- ☐ Document your (short) answer to the question above in the file *Exercise6-4.txt*

Submission: Submit the document containing your answer.

General 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 project including all relevant additional files as specified via packing in a Zip archive. Adhere to the naming convention for projects: SQE##-*Lastname_Title* (## = assignment number, *Lastname* = your name, *Title* = exercise title).
- Upload the archive at the specified submission link on the elearning platform http://elearning.fh-hagenberg.at/ until the specified date and time.