Lab work Software Testing*

1 Introduction

1.1 Objectives

The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice.

You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing. You will get a working Pacman without monsters for free, and part of the task consists of extending Pacman with moving monsters.

Tools you will learn how to use include ant for automating the build process, Java assert statements for developing built-in tests and applying design-by-contract, `JUnit` for running Java unit tests, and Emma for running test coverage tools.

1.2 Teaching Assistants

The lab will be supervised by Bart Van Rompaey (bart.vanrompaey2@ua.ac.be) and Jan Vlegels (jan.vlegels@ua.ac.be).

1.3 Approach

The work in the labs is mostly self-study. The handouts contain a chain of tasks, some more practical, others in the form of more philosophical questions reflecting on previous tasks. You are free to work in groups. The schedule, handouts and other practical information can be found at `http://www.lore.ua.ac.be/Teaching/STestenMaster/`. Part of the oral exam covers the lab sessions, focussing on the motivation to use of particular test techniques and strategies.

*Based upon a software testing course thaught by Arie van Deursen at the TU Delft.

Programming exercises are in Java. For your Java development, you can use your favorite tool suite (such as Eclipse or plain old Emacs). A distribution of the Pacman software will be handed out. The zip not only contains the Java implementation, build files, and the test suite, but also additional documentation (in the doc dir), including:

- pacman-requirements.txt: A text file describing the JPACMAN use cases
- pacman-design.txt: A text file describing the key JPACMAN design decisions.

2 Lab 1: Java Testing Tools Preliminaries

The purpose of the first exercises is to help you master your testing tools a prerequisite before you can apply systematic testing techniques to JPACMAN.

2.1 Ant

Information and documentation is available via `http://ant.apache.org/`. You should familiarize yourself with the ant build.xml provided. Experiment by trying a couple of targets, such as compile, check, clean, etc.

Exercise 1 *Add a doc target to invoke javadoc. Explain which options of the javadoc ant task you used. N.B.: the Java code is written in Java 1.5 which you have to indicate to javadoc. See the ant manual for further details.*

2.2 JUnit

Information and documentation on JUnit is available via `http://www.junit.org/`. To familiarize yourself with the way JUnit is used in JPACMAN, take a look at the various available test classes. The Java source code is in the directory *src/main*, while the test cases are in the directory *src/test* (following the directory structure as used by default in Apache Maven projects). Observe that the package structure of these two directories is exactly the same (allowing test cases to access package visible members).

All JUnit tests are invoked via the top-level *jpacman.TestAll* class. This test class is invoked via the *ant check* target. While extending and testing JPACMAN, make sure that you invoke *ant check* before and after each change you make!!

If you're not familiar with JUnit, carefully study the article "Test Infected, Programmers Love Writing Tests", by Gamma and Beck, available from the JUnit web site.

Next, consider the method adjacent for the class Cell in JPACMAN with the following responsibility:

```
/**
   * Determine if the other cell is an immediate
   * neighbour of the current cell.
   * @return true iff the other cell is immediately adjacent.
   */
 public boolean adjacent(Cell otherCell) {
    ...
 }
```

Exercise 2 *First, generate as many functional (also called responsibility-driven) test cases as you think are nesseary. Describe each test case.*

Exercise 3 *Turn your test cases into JUnit test cases in CellTest, and include an empty method body in Cell to make sure your code compiles. What happens if you run these tests?*

Exercise 4 *Write a proper implementation of adjacent and rerun your test cases.*

2.3 Assertions

To familiarize yourself with programming with assertions carry out the following steps:

Exercise 5 *Analyze the various uses of assertions (also see Binder p818). Search for assertions that are used as precondition, postcondition, and as class invariant. List one example for each category.*

Exercise 6 *Explain the differences between the JUnit collection of assert methods and the Java assert statement.*

Exercise 7 *To get a feeling of what happens when an assertion fails, include an assertion (with documentation string) that you know will fail on a point that you know will be executed by one of the tests. Run the tests and explain what happens.*

Exercise 8 *Now modify the ant build file so that the tests are run with assertion checking disabled. Rerun, and see what happens. Describe the ant file modifications you made, and describe what happens if you run the tests this way. Make your conclusions about asserts: when do you use the Java assert statement and when a testing framework?*

Finally, undo you changes to the build file, rerun to check that the assertions indeed fail, and remove the failing assertion.

2.4 Code Coverage with Emma

Information and documentation concerning the open source coverage tool Emma is available from `http://emma.sourceforge.net/`. The code coverage analyzer Emma is tightly integrated with Ant. Emma can either be invoked directly from the command line, or via the ant build file. There is also a separate version for Eclipse, available as Eclemma[1].

To run the test suite under Emma coverage, a special *ant target echeck* has been included in build.xml. This target makes use of Emma's on the fly instrumentation mode. You can invoke it with

```
ant clean echeck
```

[1] `http://www.eclemma.org/`

Furthermore, a target called *erun* to run JPacman under emma is provided as well, which can be invoked through

```
ant clean erun
```

We will first analyze how we can analyze code covered when just executing the game. To that end, run the pacman game interactively using

```
ant clean erun
```

Start the game, make a couple of moves and exit the game. Next, inspect the coverage results, by opening the file `target/coverage/coverage.html` in your web browser.

Exercise 9 *Navigate through the coverage results by clicking on packages or classes. List the three most interesting percentages you found, and explain them. Now start the game again under Emma coverage, and try to play the game in as many different ways as possible, in order to try to execute all code. Analyze the coverage data in the generated html files.*

Exercise 10 *Were there parts of the code you did not hit? Explain why you were not able to hit them and why this makes sense. Next, we will study how Emma can report coverage of test cases, which can be computed by:*

```
ant clean echeck
```

Exercise 11 *Again, inspect the coverage results. List the three most interesting percentages you found, and explain them. Finally, we will study how Emma handles asserts.*

Exercise 12 *What color are most of the assert statements? Why? How does this affect the percentages provided by Emma?*

Make sure that you understand the precisely definition of Emma's coverage measures. Compare with alternative coverage measures that you are familiar with from literature or tool practice.

2.5 Stubs and Mocks

Read the article by Martin Fowler about the difference between stubs and mocks[2]. Familiarize yourself with a mock library for Java such as jMock (`http://www.jmock.org`) or EasyMock (`http://www.easymock.org/`).

Exercise 13 *Reconsider some of the functional tests you wrote in Exercise 2. Write them now using mocks. Use a mock library of your choice.*

[2] `http://www.martinfowler.com/articles/mocksArentStubs.html`

Exercise 14 *Is the coverage resulting from the mock-based test the same as the original? Compare both approaches. When would you prefer mock testing?*

2.6 Testing Frameworks

JUnit is not the only (unit) testing framework for Java. Check out the tools JTiger[3], TestNG[4] and JExample[5].

Exercise 15 *How do they compare to JUnit? Which features do you consider useful in case of JPACMAN/in case of systems with other characteristics?*

[3] http://jtiger.org/
[4] http://testng.org/doc/
[5] http://smallwiki.unibe.ch/jexample/