University of Toronto - Department of Computer Science

CSC410, Fall 2016

Assignment 2

This assignment is worth 15% of your final mark
Due: Tuesday, December 6, 2016 at 23:59
Please indicate your submission as "final" if you don't intend to overwrite it with late submissions.

Instructions

The objective of this assignment is to help you learn to apply the various techniques and test strategies discussed during the lectures in practice.

You will apply these techniques to a simple game called JPACMAN implemented in Java. The amount of coding that needs to be done in this assignment is relatively small; the focus is on testing. You will get a JPACMAN lite version missing some functionality, and part of the assignment consists of extending it appropriately.

Tools that you will learn to use include **Maven** for automating the build process, **JUnit** for running Java unit tests, **EclEmma/Jacoco** for computing test coverage, **Symbolic Pathfinder (SPF)** for symbolic execution and test case generation, and **PIT** for mutation testing.

The assignment is to be done in groups of two. Submit a report '**as2.pdf**' answering exercise questions as well as describing your implementation and testing efforts, and a compressed archive containing your modified project '**as2-project.zip**' to Markus.

Questions? Ask them on Piazza (folder 'assignment2').

Both of the group members should sign this document.

==========

CSC410, Fall 2016 - Assignment 2
Name: _____
Student Number: _____
Lecture: o Monday o Tuesday
Name: _____
Student Number: _____
Lecture: o Monday o Tuesday
We are the sole authors of this homework. Signatures: _____

==========

Part 0: Familiarize with the Tools and Techniques [0 marks]

1. Download and set up Apache Maven: https://maven.apache.org/
    a. Maven is already available on CDF machines, but it is fairly easy to set up on your own machine.
    b. Download and unzip maven, configure `PATH` variables.
    c. On CDF, you need to explicitly switch to Java 1.8 for building JPACMAN: `setenv JAVA_HOME /local/packages/jdk1.8.0/`
    d. Try running a Junit test with Maven (in the root directory of JPACMAN): `mvn test -Dtest=LauncherSmokeTest`
2. Measure code coverage with Jacoco: http://www.eclemma.org/jacoco/
    a. Jacoco is already integrated in the build file `pom.xml`.
    b. Run some tests: `mvn test -Dtest=LauncherSmokeTest`
    c. Use the Jacoco coverage plugin for Maven to generate coverage report (in 'target/site/jacoco'): `mvn jacoco:report`
3. Set up and use PIT: http://pitest.org/
    a. Follow the quick start guide.
    b. Modify '`pom.xml`' to include PIT plugin into the build.
    c. In the root directory of JPACMAN, run mutation testing using: `mvn org.pitest:pitest-maven:mutationCoverage`
4. Download and set up SPF: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
    a. SPF is already available on CDF machines (recall from tutorial 3).
    b. Try the provided example in: `/u/csc410h/fall/pub/tutorial3/`
5. (Optional) Install a recent version of Eclipse which has m2e maven support.
    a. Import the JPACMAN project into Eclipse.
    b. Try running a Junit test from Eclipse.
6. (Optional) Install and use EclEmma Eclipse plugin: http://www.eclemma.org/
    a. Launch the game with the EclEmma coverage mode.
    b. Read the coverage report produced while testing.

Part 1: Assess Quality of Tests using Line Coverage [25 marks]

1. Do some manual exploratory testing of the game by activating various functions and exercising various control sequences (e.g., clicking buttons, keystrokes, etc.). Report three different scenarios you have tested and the corresponding coverage results. Explain the differences, if found any. Describe your test scenarios in detail so that anyone can reproduce your results.
2. Measure the code coverage of the existing test suites as a whole in "`src/test/java`". Report the coverage percentage. Identify the three least covered application classes (in "`src/main/java`"). Explain why the tests for them are adequate or how they can be improved.

3. Measure the code coverage again, but this time with a configuration that has runtime assertion enabled (add '`-ea`' as VM argument). To do this, right click on the project, select "Coverage As", then go to "Coverage Configurations". Then under "Arguments" add "`-ea`" to VM arguments. Explain the coverage changes you see.

Part 2: Assess Quality of Tests using Mutation Testing [25 marks]

1. Run PIT with the default set of mutation operators on the existing test suite and measure mutation coverage. Report the results and compare them with line coverage results you got earlier. Explain what you see.
2. Run PIT with only "Conditionals Boundary Mutator", "Increments Mutator", and "Math Mutator", respectively. Compare the results and explain.
3. Add one Junit test case to an appropriate package (e.g., in "`src/test/java/...`") so that with it more mutants can be killed using default PIT configurations (i.e., the mutation score increases). Include your test case in the report and explain.

Part 3: Extend Test Suite with Symbolic Execution [40 marks]

1. Use SPF to generate test cases for the '`withinBorders(int,int)`' method in the '`nl.tudelft.jpacman.board.Board`' class. Note that in order to apply SPF on a non-static method, you need to write a driver method '`static void main(String[])`' which instantiates the target class and calls the target methods. Write a fully functional JUnit test class '`JPFBoardTest`' for the test cases generated in a new file '`/src/test/java/nl/tudelft/jpacman/board/JPFBoardTest.java`'. The test class should have a test fixture (i.e., @Before) and several test cases (i.e., @Test). Include your test class in the report as well.
2. Does the generated test suite cover all boundary values (i.e., in, off, and on boundary)? Explain. Complete the test suite if you find it to be incomplete.

Part 4: Implement a New Feature and Test it! [60 marks]

1. Implement the following feature and apply testing strategies you learned during lectures to build a good test suite for it:

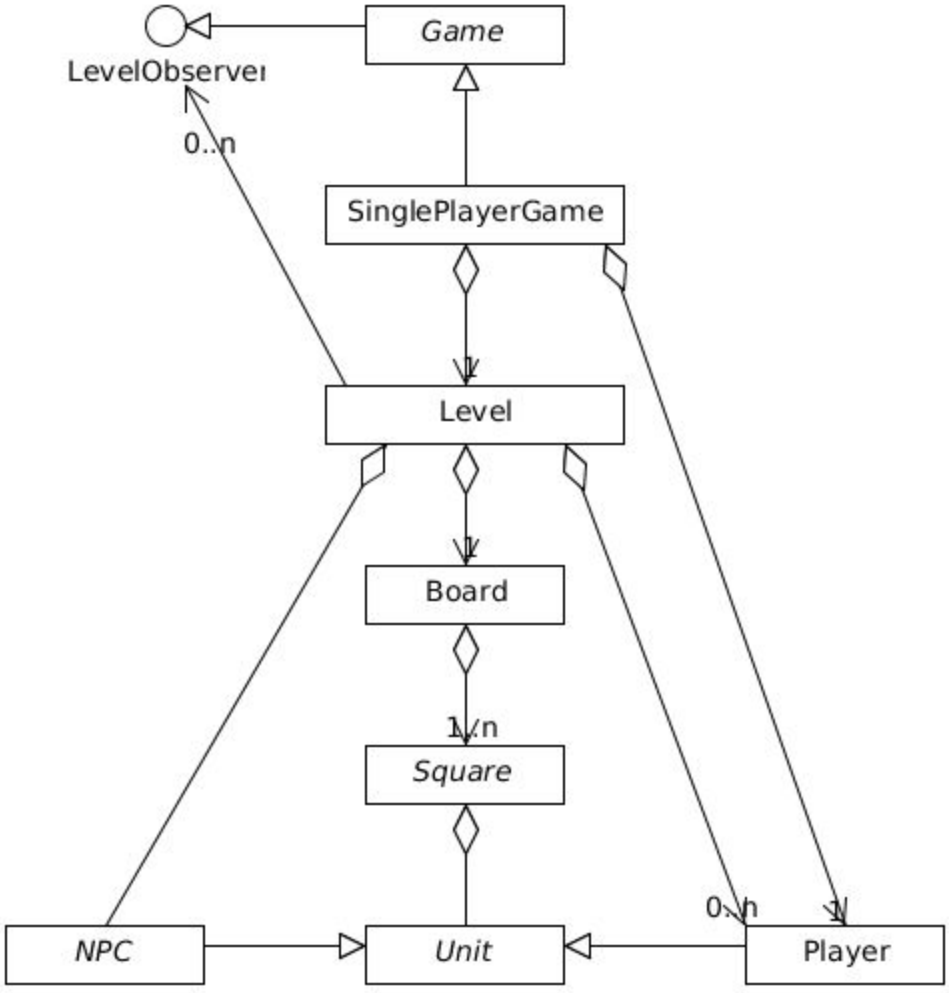
   A "Freeze (Unfreeze)" button which stops all Non-player Characters (NPCs) when first clicked and resumes them in action when clicked again. Everything else including the player's action, pellets, score board, etc., should not be affected.

2. Was you test suite able to capture any bugs in your implementation?
3. Document your testing efforts with a test plan (refer to PY'07 Page 463 for an example). What features have you tested? What approaches have you used to improve your tests? The marks will be given based on both your understanding as well as applications of different test strategies and the quality of your implementation and tests.

Appendices

UML diagrams are provided for you to understand the high-level design of the JPACMAN framework. Knowledge about UML is optional.

1. UML diagram for a single player game.



2. UML diagram for factory wiring.