Software Quality and Testing

SQT Labwork, CSE1110

Edition 2018/2019

Arie van Deursen, Maurício Aniche
Casper Boone, Max Lopes Cunha, Azqa Nadeem

Delft University of Technology

June 4, 2019

1 Part II - Structural testing and Mocking

1.1 Coverage Analysis in IntelliJ

For coverage analysis to work, make sure you have setup your IDE precisely as instructed before (check the instructions of previous assignments, if not). To run the IntelliJ coverage with branch coverage enabled, click on your run configuration, then ”Edit Configurations”. In the tab ”Code Coverage”, switch the mode from ”Sampling” to ”Tracing”. Save the configuration with ”Apply” and run it. Take note that if branch coverage does not show up, double-check the run configuration again. You might only have altered the run configuration for a different class.

- Run the tests ”with Coverage”.
- Inspect the coverage: Double clicking the `nl, tudelft, jpacman` packages will lead you to the actual coverage results.
- Find the package with the lowest coverage, and inspect some of its classes.

1.2 Mock Objects

When testing a given class, mocks can be used to control the behavior of classes your class-under-test depends on. They act as an object that substitutes your original class.

Mocks increase both *observability* and *controllability*. Observability is the measure of how well you can measure the internal state of an object. If an object is not observable (not to be confused with the Observer/Observable pattern!) then testing is more difficult. Controllability on the other hand, is the measure of how well we can *manipulate* the internal state. Mocks let us do both; we can observe which methods were called with which arguments and what they returned.

Don’t forget to use separate commits for each exercise and reference which exercise you worked on in the commit message.

Exercise 1

- *Write a test suite for the `level.MapParser` class. Start out with the* nice weather behavior*, in which the board contains expected characters. Use Mockito to mock the factories, and use Mockito to verify that reading a map leads to the proper interactions with those factories.*

Exercise 2

- *Extend the test suite to bad weather situations. These bad weather cases will force the class to raise the proper exceptions.*
  It is quite easy to think about bad weather situations here, but clearly the ones that are more likely to happen are more important. Think about your previous programming assignments: what kind of mistakes have you had when reading data from a file?

1.3 Branch Coverage with Mocks

Next, we will add test cases that achieve *branch/decision coverage* for the method `game.Game.start()`.

The challenge in this method is to ensure sufficient *controllability* and *observability*, which is why will resort to mock objects.

Create a `GameUnitTest` class, which defines a `SinglePlayerGame` with mocked dependencies. The different test cases can then mock the dependencies in different ways, in order to achieve branch coverage.

Exercise 3

- *Write test methods that together achieve 100% branch coverage of the if-statements of the `Game.start()` method. Write down the coverage you achieved in your report.*

1.4 Testing Collisions

In this exercise we will look at collisions between the player, ghosts, and pellets. Collisions can happen between *units*: To see which units are implemented inspect the type hierarchy of the `board.Unit` class.

Exercise 4

- *Analyze requirements (found in doc/scenarios.md) and derive a decision table for the JPacman collisions from it. In this decision table you should encode the outcomes of collisions between two pairs of entities. You are free to filter out collisions that do not occur, such as two `Pellet`'s colliding. To give you an idea, look at the table below. Note that this table is incomplete and may have too many or too few columns.*

Collider	??	??	??	Ghost	??	??
Collidee	??	??	Pellet	??	??	Player
Consequence	??	??	??	??	??	??

Exercise 5

- *Based on the decision table for collisions, derive a JUnit test suite for the `level.PlayerCollisions` class. You should be as rigorous as possible here; think not only of collisions that result in something, but also on collisions where "nothing happens".*

  Hint: Use mocks.

- The `PlayerCollision` class is far from ideal, as it does not scale well to more realistic collision maps. An alternative is the (reflection-based) `DefaultPlayerInteractionMap`, which makes use of the (more complicated) `CollisionInteractionMap`

Exercise 6 *Restructure your test suite from exercise 5 so that you can execute the same test suite on both `PlayerCollision` and `DefaultPlayerInteractionMap` objects.*

Hint: Use a parallel class hierarchy for your tests.

Exercise 7

- *Analyze the increase in coverage compared to the original tests we gave you at the beginning, and discuss what collision functionality you have covered additionally, and which (if any) collision functionality is still unchecked.*

1.5 Pragmatic testing

Exercise 8

- *See the `Ghost#randomMove()` method. It makes use of Java's `Random` class to generate random numbers. How would you test such method, if everytime you execute the method you get a different answer? Explain your idea (max 100 words)*

Exercise 9

- *JPacman contains a test that can become a* flaky test*: see LauncherSmokeTest.smokeTest. Read the test and find out why this test can be flaky. Next, discuss other reasons why a test can become flaky and what can we do to avoid them.(max 100 words)*

  You can read: `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html`.

Exercise 10

- *What is your opinion regarding achieving 100% of code coverage? What are the advantages? What are the disadvantages? How should one deal with such metrics, in your opinion?(max 100 words)*

Exercise 11

- *You made intensive use of mocks in this assignment. So, you definitely know its advantages. But, in your opinion, what are the main* disadvantages *of such approach? Explain your reasons. (max 100 words)*

  You can read `https://8thlight.com/blog/uncle-bob/2014/05/10/WhenToMock.html` and `http://www.jmock.org/oopsla2004.pdf`.

Exercise 12

- *Our test suite is pretty fast. However, the more a test suite grows the more time it takes to execute. Can you think of scenarios (more than one) that can lead a single test (and eventually the entire test suite) to become slow? What can we do to mitigate the issue?(max 100 words)*

Exercise 13

- *There are occasions in which we should use the class' concrete implementation and not mock it. In what cases should one mock a class? In what cases should one not mock a class?*

  Hint: Think about the test level (unit, integration, system testing). You can also read the following paper, if you are curious about how mock objects evolve over time: `https://bit.ly/2HMVHGH`.

1.6 Security Testing

Java offers a "plug-and-play infrastructure" by allowing us to dynamically load components. We do not need the source code for these components to run – the .class or .jar file are dynamically loaded in the code for the added functionality. In this case, the score calculator of JPacman is one such dynamically loaded component. In the default version of JPacman, the points are calculated using `DefaultPointCalculator`.

We have received another plugin for score calculation, called the `AmazingPointCalculator`. Our vendor claims that this plugin is meant for more competitive users as it is more difficult to earn scores. You can see `AmazingPointCalculator.class` in `/resources/scoreplugins/` directory. Go ahead and enable it by editing `/resources/scorecalc.properties` and replacing `DefaultPointCalculator` with `AmazingPointCalculator`. Play the game as usual.

Exercise 14 *Have you noticed any weird behavior in the game? Inspect by playing the game a number of times and report what you observe.*
Hint: There are 4 anomalous behaviors associated to the score and the direction.

The new plugin seems to do a lot more than calculating scores. But how can we find everything that it does? Security testing can be done in two ways: (i) Statically analyzing the code base for potential vulnerabilities, (ii) Dynamically running the code in order to observe anomalous/illegal behaviors. Since JPacman uses dynamically loaded plugins, statically analysing the code will not help since the code base for plugins is not available.

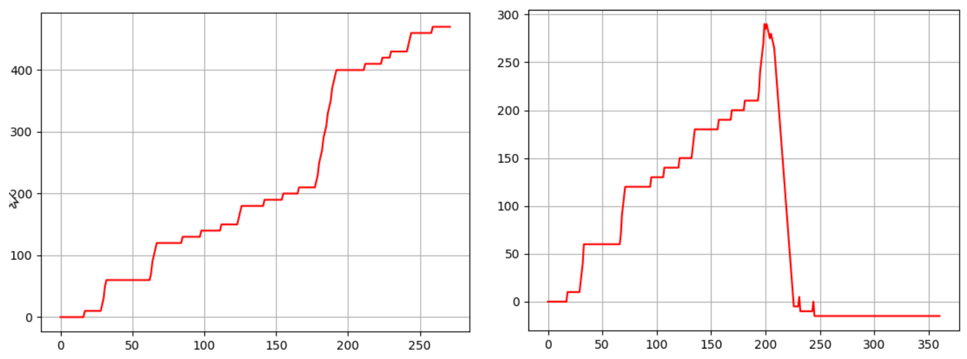
Executing the application, observing its behavior and comparing with known-normal behavior is another way to do security testing. One thing that really helps in keeping track of the behavior is logging. In this assignment, you will run JPacman to generate logs associated to different aspects of the game. You will use these logs to build graphs and compare the behaviors of the two score calculator plugins. The result should look somewhat like Figure 1. The two plots show how the score changes overtime for a player using the two scoring plugins. We can see that there is visibly something wrong with the second plugin as the score suddenly drops after the 200^{th} iteration.

To help you not having to play JPacman an endless number of times, we have provided a fuzzer that automatically plays the game on your behalf. Fuzzing is an automated testing technique that utilizes all possible mutations of test inputs in order to find crashes. In this scenario, we will use a naive fuzzer to test all input mutations with the purpose of generating logs.

The fuzzer is implemented as a test class in `fuzzer/JPacmanFuzzer.java` under `default-test/`. You can enable `fuzzerTest(RepetitionInfo)` to include it in the test suite. At every iteration, it randomly picks a direction to move until Pacman dies. Interesting variables are written to a log file after each move. You can change `RUNS` to the number of times you want to re-run JPacman. Each instance will have a log file of its own. The log files are located under `behavioral-analysis/logs/` directory. Within the `behavioral-analysis/` directory, you will also find a simple Python script, `plotLogs.py`, that you can use to plot the log files. Usage instructions are in the README file.

Exercise 15 *Run the fuzzer multiple times and plot the generated logs. Perform this for both Default and Amazing Point Calculators, and compare their logs. Report what differences you see in their behaviors. The solution should contain snapshots of both normal and abnormal behavior, followed by an explanation of what behavior you observed and which condition triggered it. Find all 4 anomalous behaviors in this way.*

Exercise 16 *Find the source code where the PointCalculator is dynamically loaded. Next, run the static analysis tools that come with JPacMan repository by running*



(a) Scorer 1 (b) Scorer 2

Figure 1: Behavior comparison of score calculators

`./gradlew staticAnalysis`. Do you see any security warnings associated to this piece of code reported by SpotBugs? Determine why (or why doesn't) SpotBugs give a security warning. Go through the OWASP Top 10 vulnerability list and determine which one applies to that piece of code. Briefly reflect on it.

Exercise 17 *How can the security problem(s) associated to dynamic class loading be fixed? Briefly explain at least three different possibilities. (max 100 words)*

1.7 Submit Part II

- In your report, analyze whether the code is ready for submission. Explain or eliminate checkstyle or SpotBugs violations that remain (if any) and include a brief assessment of the additional adequacy achieved in JPacman thanks to your new classes. In addition, reflect on your continuous integration server results, your commit behavior, and the new knowledge acquired.
- Commit and push all changes, prepare your final report as pdf, and upload your pdf as part of your submission of release "Part II" in GitLab.