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

April 24, 2019

1 Introduction

In this document, you will find everything about your labwork.

You will apply the different testing techniques we teach to a simple game called JPACMAN, inspired by Pacman and written in Java. The amount of coding that needs to be done is relatively small: the focus is on testing.

The labwork is divided into four parts:

- Part 0: Get acquainted with the environment and tools.
- Part 1: Unit tests and boundary tests.
- Part 2: Structural testing and mock objects.
- Part 3: System tests, state-based testing, and mocking.

2 Part 0: Prerequisites

Before starting on the exercises that earn you points, the prerequisite steps guide you through the tools and processes to be used.

Read all remarks carefully, even if you have experience with some of the tools. Some of the fine print details are important and specific to the JPacman setting.

2.1 Register on Brightspace

To facilitate the forming of groups, we make use of `https://brightspace.tudelft.nl/`

- Log into Brightspace using your own NetID and password.
- Navigate to the CSE1110 Software Quality and Testing course.
- In the top menu, navigate to Collaboration, and then Groups
- Click on View Available Groups, and use the "Join Group" link to join a group of your choice.
- Make sure your partner joins the same group. If you have joined the wrong group by mistake, contact a TA to help you out.

Once groups have been formed, we will use these groups to create accounts for you on GitLab. For more information about GitLab, see the next section.

2.2 GitLab

To facilitate git and continuous integration, we make use of `https://gitlab.ewi.tudelft.nl/`. In order for you to use this service, a few steps need to be taken.

- You should already have a GitLab account. If you do not have one, let our head TAs know.
- You should also have a SSH key configured already. If not, create a SSH key following the instructions: `https://gitlab.ewi.tudelft.nl/help/ssh/README#generating-a-new-ssh-key-pair`.

  We know that for some Windows users, configuring the SSH key can be tricky. You can clone your repository using the HTTPS link, which will not ask for the SSH key (although it may ask you for your GitLab password).
- We will create a repository for you to work on JPACMAN. You can find it in *Projects → YOUR-GROUP-NAME / jpacman*.
- You will see a git clone URL that you can use to clone the project (it should start with *git@gitlab.ewi.tudelft.nl:*), as well as a list of commits currently in your repository. Make sure you use the **git URL**, and not the HTTPS one.

2.3 Revision control: git

For this course, we'll mostly use the `clone`, `add`, `commit`, `push`, `pull`, `branch`, and `checkout` commands from `git`.

If you need to learn git, you can find a hands-on tutorial provided by GitHub here: `https://try.github.io/`. Later, if you want to become proficient in Git, you can dive into the a more complete book on the topic (`http://git-scm.com/book`).

Making a good use of Git is something that developers really appreaciate in real life. Thus, we expect you to also follow some best practices:

- Make sure you commit frequently: *Commit at least once, but possibly more often per exercise!*
- Write clear commit messages: In a collaborative environment, it is incredibly important to give others the chance to become aware of your changes. Commit messages serve as a summary of these changes.[1] *Your commit messages should thus always mention the exercise you were working on!*
- Push frequently to the GitLab remote so that the automatic build is triggered.
- Use branches: You'll get most benefit from git if you also use branches, for example one branch for every question, or for every group of questions that belong together.
- Use merge requests as supported by GitLab, and avoid committing to master directly.[2]
- Use both accounts: Each of the two students in a team should commit around half of the changes. Collaborating and working from one computer is fine, but both students should be equal and take turns (e.g. after every section in the assignment) in actually committing.

The way you use Git is part of the grade. Do your best.

We also suggest you to use the command line. However, if you are still struggling with Git, using a visual interface can help you in understanding what's going on under the hood. Feel free to use tools such as Fork[3].

However, please note that TAs may not be familiar with the specific tool you chose to use. It is therefore recommended that you *also* install a command-line client (in case of Windows, we recommend Git Bash).

2.4 IDE: IntelliJ

We will use IntelliJ in this course. We suggest you to use the latest version (2019.1), as it better provides support to JUnit 5.

[1]`https://chris.beams.io/posts/git-commit/` is a great guide on how to create commit messages.
[2]`https://zef.me/never-commit-to-master-d1d8b3bde6c6`
[3]https://git-fork.com/

You will need the ultimate edition that supports coverage analysis. An academic license offering this is available through JetBrains[4], using your @student.tudelft.nl e-mail address.

2.5 Getting started with JPACMAN

- Clone JPACMAN using the git repository URL you find in your GitLab project.
- In your IDE, choose "Import Project". Choose the folder that contains the cloned JPACMAN project. IntelliJ will automatically highlight Gradle. Keep clicking on Next until it's done.

  There's usually no configuration to be done; however, in some machines, IntelliJ doesn't find the path where Gradle is actually installed. In that case select "Use gradle 'wrapper' task configuration" instead of "Use default gradle wrapper".

  After importing, you might see a popup message "The IDE modules below were removed by the Gradle import: jpacman". You can safely ignore this message.
- Let's run a smoke test, just to make sure everything is working. Open the `LauncherSmokeTest` class. It is under *src/default-test/java/nl.tudelft.jpacman*. Right-click the source code in the code editor, and choose `Run LaunchSomeTest` button. You should quickly see JPacman opening up and closing again.
- Note that we have *two* test folders. One is called *default-test* (src/default-test) and the other one is just called *test* (src/test). Although this is not common in most projects, we separate the tests we give you from the very beginning (the 'default tests') and the tests you will write by yourself. All **your** tests should be written in the **src/test** folder. Please, follow this guideline. This will help us in assessing you later. Tests in the wrong location will be not be graded.
- Do a small change and see it happening in JPACMAN. For example, open the `PacManUI` class that is under the `nl.tudelft.tudelft.jpacman.ui` package. The first line inside the constructor is something like `super("JPacman");`. Change it to `super("JPacman <current year>");`.
- Run the `LauncherSmokeTest` again and see the new title of the window!
- Commit and git push to the GitLab remote (origin). Remember to write a proper commit message.
- Log in to your GitLab account, open your project's dashboard. Inspect what you see.

2.6 Using Gradle via command-line

Gradle is used for configuring the project, its build and test process, and its dependencies. While IntelliJ has a built-in version, you should install Gradle yourself (see `https://gradle.org/install/`) if you want to use the command-line.

[4]`https://www.jetbrains.com/student/`

- Run the tests and static analysis tools via gradle (`gradle check`). If you only want to run the tests, you can use `gradle test`. If you only want to run the static analysis tools, you can use `gradle staticAnalysis`.
- `gradle check` will fail as soon as the tests or one of the tools fail. However, you can also inspect the generated reports at `build/reports`.[5] Note that it contains many folders, one for each tool we run: checkstyle, jacoco, pmd, spotbugs, and tests. Identify, e.g., the test results and the coverage data (which we will study in depth later on in this course).
- Inspect the `build.gradle` file, which is where most of this magic happens. Gradle is a well-known build automation tool; get to know it!

2.7 Our First Tests

- Open the `board.DirectionTest`.

Exercise 1

- *Just to get you started, create additional test methods in `DirectionTest` for e.g., the south, east, and west directions.*
- Run the tests, and ensure they pass.
- Commit, and git push to the GitLab remote (origin).
- Log in to your GitLab account, open your project's dashboard. Inspect what you see.
- In your IDE, modify one or more of your test cases so that they fail. Commit in git, and push to GitLab. After a while the GitLab integration server will rebuild automatically, and should send you an email about a failed test. Go to GitLab and see what a broken build looks like.
- Repair the tests so that they pass on your local computer again, commit and push.[6]

Let's test some more complicated behavior.

- Take a look at the `OccupantTest` class. It tests whether `Units` correctly (de)occupy squares.

Exercise 2

- *We left three test methods empty there for you. Your goal is to implement these tests.*
  1. The `noStartSquare()` test asserts that a unit has no square to start with, i.e., a unit "has no square" at the beginning.

[5]You might have to refresh the `build` folder in your IDE.

[6]An alternative way to do this in git is to use the `git revert` command. Feel free to use more advanced Git commands if you know them.

2. The `testOccupy()` verifies that the unit indeed has the target square as its base after occupation. In other words, if a unit is occupied by a(ny) basic square, then one should contain the other.
3. Finally, the `testReoccupy()` verifies that the unit indeed has the target square as its base after double occupation. What happens if the unit is reoccupied by another square?

- Make sure all your tests are passing.
- Commit and push your tests.

2.8 Assertions

JPacman contains over 50 `assert` statements. This statement raises an exception if its Boolean condition is false.

The assertions are part of the code in order to document *preconditions*, *postconditions*, and *invariants*, as used in "Design by Contract". We'll study more about it later on.

Runtime assertion checking can be enabled or disabled through the `-ea` (or `-enableassertions`) option passed to the Java Virtual Machine (JVM). The default behavior of both `gradle` and IntelliJ[7] is that assertions are *enabled during testing* (e.g. when running `gradle check`), and *disabled in production* (e.g., when running `Launcher.main`).

- Browse through the JPacman code to explore the use of assertions. As an example, take a look at the `nl.tudelft.jpacman.board.Board` class.
- In the Board class, look at the `invariant` method. What does it check?
- Create a test class for `Board` in which you construct a board with a correct (1×1 is large enough) grid, with a correct `BasicSquare` on it. Run your test in either the IDE or in gradle. Since the board is valid your test should pass.
- Create a second test case in which you construct a similar board, but with just one null square. For both, write a test for the `squareAt` method. With what exception does your test fail?
- Now in your IDE, look at the run configuration for the `BoardTest` class you created (Run / Run... / Edit Configurations). Find the `-ea` option, remove it, and re-run the test. What happens?
- Next re-add the `-ea` option to your Run Configuration, just in case you want to rerun that test case again.
- Commit your `BoardTest`, and push it to GitLab. The build should fail in this case. Is assertion checking enabled or disabled on GitLab? How do you know?
- Next, remove the failing test (that caused the invariant to be violated), and commit and push so that you have a green build again.

[7] By default, Eclipse has assertions *disabled* in tests, another reason why in this course we will use IntelliJ.

2.9 Static Analysis: Checkstyle, PMD, and Spotbugs

We're using three different tools for statically analyzing the quality of the code: Checkstyle, PMD, and SpotBugs.

Any warnings triggered by these tools should be either resolved, or suppressed together with an explanation in your report for the rare cases that you want to keep the code as is despite the warning.

You can always check the results of these tools in GitLab's continuous integration, or by running `gradle check`.

- If these tools find any problems, fix them.
- Install the IntelliJ Checkstyle and PMD plugins (`Preferences ... / Plugins`), and that will show you the feedback right in the IDE. The Checkstyle plugin will be run automatically, while you have to run the PMD plugin manually.

  There is no Spotbugs plug-in for IntelliJ yet. For now, you can only run it via Gradle.
- Configure checkstyle so that it uses JPacman's `checkstyle.xml` as rule set: `Preferences ... / Other Settings / Checkstyle / "+"`, then pick the `checkstyle.xml` file that exists in JPacman, and tick it as the selected configuration.
- Configure PMD and `pmd-rules.xml` in the same way.
- Enable checkstyle checking for your project, and visit all violations that you encounter during development (if you just cloned, there should not be any). Adjust your code so that no warnings for your code are left.
- Do the same for PMD and SpotBugs (likely there are no warnings yet).
- Commit and push your changes.
- To understand how checkstyle works, have a look at the checkstyle.xml file. Pick three checkstyle modules listed in the .xml configuration file, lookup these modules in the online documentation, and explain their rationale and working.

If you see a rule that you'd like to modify or add, do so in the checkstyle.xml file, commit and push. However, make sure you clearly explain your reasoning to do so in your report. You lose points if you disable a rule without an explanation.

2.10 Principles of software testing

In class, we just saw some important principles of software testing. Now, use your own words to describe:

Exercise 3 *Why can't we exhaustively test our entire software project? What should we do instead? (max 100 words)*

Exercise 4 *What is the pesticide paradox about and what does it imply to software testers? (max*

100 words)

Exercise 5 *Why should we automate, as much as possible, the test execution? (max 100 words)*

You should answer all the questions in your report. This is how the next parts of the assignment will look like. You will answer the theoretical questions in your report, and do all practical exercises in your JPACMAN repository. Your TAs and colleagues will review both your report and source code.

2.11 Upload your Solution to GitLab and Peer

- Run `gradle check` to generate all reports, inspect and fix all open warnings.
- Commit and push to the GitLab git remote.
- On GitLab, check your build status, and verify that everything is OK.
- On GitLab, go to the "Repository → Tags" page of your project. Tag your last commit as your *Prerequisites* assignment.
- Upload your report as a PDF. For this, use the *Attach File* link under the *Release notes* input.
- Upload a zip file containing your report and your JPacman source code to Peer. Remember that this zip must be anonymised (i.e., no names and group numbers in there).
- *Congratulations, you have uploaded your first release.*

In the rest of the labwork, make sure you work in small steps; commit often; push often; and listen to the feedback provided by the GitLab continuous integration system.

Show us you're a team by taking turns in submitting merge requests, reviewing and merging them.

Your use of git and GitLab will be evaluated as part of your score, so try to make optimal use of the infrastructure offered.

3 Part I: Unit tests + Boundary Tests

3.1 Smoke Testing

Test cases can target different *levels*, ranging from units (individual methods) to the complete system. We will start by looking at a system-level test that tries to exercise the system from *end-to-end*.

We call this one a smoke test. A *Smoke Test* is a simple system test that exercises minimal system behavior. If we just start the system, and we "see smoke" (i.e., not even the smoke test passes), there is no point in moving to the next step of the software development cycle. JPACMAN has a simple smoke test already: the `LauncherSmokeTest`.

Exercise 6 *Execute the smoke test, with coverage enabled. Name 2 classes that are not well-tested, and explain why the smoke test does not cover it.*

Exercise 7 *Have a look at class `game.Game`, method `move`. Is it covered by our smoke test?*

Now, comment out the last line, invoking the `move` method in the actual Level. Run the smoke test again. Explain the failure you see, and explain to what extent you think the resulting test failure can be helpful in fixing the system.

Exercise 8 *Undo the previous change, but now change `board.Direction.getDeltaX` so that it returns `dy` instead of `dx`.*

Next, undo the to changes you made, and ensure all tests pass again. Explain what you see. Was it easy to understand where the problem is?

Working on a code base that was written by others is probably what you will face in industry. Use this smoke test (which exercises the most important parts of the system) as an excuse to explore and understand the source code of JPacman.

Exercise 9 *Then, provide at most two paragraphs explaining how `Game`, `Unit`, `Board`, and `Level` classes are related to each other.*

3.2 Unit Testing

Exercise 10 *It is time to focus on unit testing. Our goal is to make sure that the units of our system work, at least when tested in isolation. But, before diving into unit testing, we should define what a unit represents. There are many ways to define it in any software system*[8].

Osherove[9] defines it as *"A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work. [...] A unit of work is a single logical functional use case in the system that can be invoked by some public interface (in most cases). A unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified."*

[8] Jorgensen, P.C., 2016. Software testing: a craftsman's approach. CRC press.

[9] Osherove, R., 2015. The art of unit testing. MITP-Verlags GmbH & Co. KG. Vancouver.

JPacman has important behavior that is currently not tested, and a unit test fits: the movement of the ghosts.

1. Check *Clyde* (in the `nl.tudelft.jpacman.npc.ghost` package) and read its javadoc to understand how this ghost works.
2. Take a look at the `Optional<Direction> nextAiMove()` method. Think about test cases to make sure Clyde works as expected.

   Tip: There are at two good weather cases and two bad weather cases that should be tested. What are they?
3. Write at least four JUnit tests for this method in a *ClydeTest* class.

   Instantiating the *Clyde* class and adding to a map requires quite some understanding of JPacman. We give you a head start: see the *GhostMapParser* class:

   (a) Think about how to instantiate the *GhostMapParser*. What classes does it require in the constructor? How do we instantiate them?

   (b) In the tests, we need to build a level, based on a map. This map should be designed based on what we want to exercise in that test. For example, if the map is a list of strings as the following: {`"############"`, `"#P␣␣␣␣␣␣␣␣C#"`, `"############"`}, the Pacman and the Clyde are on the same row, 8 columns apart.

   (c) The provided `GhostMapParser#parseMap()` method receives the map, and returns a `Level` that uses the provided map.

   (d) If your test involves a player on the map, do not forget to register it in the level (`Level#registerPlayer`) as well as to set its direction (`Player#setDirection`).

   (e) You should assert the direction that ghost takes based on its current position in the level. For that, you need the ghost that exists in the created level. For that, we provide the `Navigation#findUnitInBoard()` method.

   Use the *GhostMapParser* to support your JUnit tests. Also, feel free to explore how the *Ghost* parent class work.
4. Add javadoc comments to each of the test methods. The documentation should explain the test case (and to which partition it belongs).

   Tip: Good naming matters. Give good names to your test methods.

Exercise 11 *There is one more ghost that can be tested: Inky.*

1. See how the Inky ghost works and, more specifically, its `Optional<Direction> nextAiMove()` method.
2. Think about partitions/cases. What tests should you design to make sure this ghost works? Read its documentation and its source code; use all the information you have in hands!

   Tip: There are at least 5 cases (2 good weather and 3 bad weather).

3. Write at least five JUnit tests for this method in a *InkyTest* class.

   You can use the *GhostMapParser* here. However, the parser is not ready for Inky. You should make sure the `addSquare()` method is also able to build Inky ghosts.

4. Add javadoc comments to each of the test methods. The documentation should explain the test case (and to which partition it belongs).

Bonus assignment (not compulsory): If you want an extra challenge, the method `squaresAheadOf` in the `Unit` class is also not tested (although you already indirectly tested it via the ghosts in the previous exercise). Write at least three tests for this method.

3.3 Boundary Testing

During the lectures, the 1×1 domain testing strategy was explained. Here we'll use it to provide a unit test for the `board.Board.withinBorders` method.

Exercise 12

- *Provide a domain matrix for the desired behavior of the boundary values in the* `withinBorders` *method.*

Exercise 13

- *Implement the corresponding test by using the JUnit 5* `ParameterizedTest` *annotation.*

3.4 Understanding your tests

Tests can be divided into three parts: *Arrange, Act, Assert (AAA)*. Let's discuss each of these parts.

Exercise 14

- *What can we do to avoid code repetition during the Arrange part of the unit test? (max 100 words)*

Exercise 15

- *Note that our tests always make use of "clean instances" of the class under test. See BoardFactoryTest as an example: the `setUp()` always instantiates a new BoardFactory instance. What are the advantages of such approach? (max 100 words)*

Exercise 16

- *JUnit and related libraries provide developers with different ways to do assertions. Some can be better than others in specific contexts. Which one is a better assertion, supposing some int `a`? 1) `assertEquals(1, a);` or 2) `assertTrue(1 == a)`? Discuss the differences between both assertions.(max 100 words)*

Exercise 17

- *Up to now, we only tested public methods. Take a look at the MapParser class: it contains many large private methods. Do we need to have specific tests for them, i.e., test them in isolation? Why? Why not? (max 100 words)*

3.5 Submit Part I

Exercise 18

- *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. Also reflect on your continuous integration server results, and your commit behavior.*

Exercise 19

- *Commit and push all changes, prepare your final report as pdf, and upload your pdf as part of your submission of release "Part I" in GitLab.*

Exercise 20

- *Upload a zip file containing your report and your entire JPacman source code to Peer. Remember that this zip must be anonymised (i.e., no names and group numbers in there).*