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 11, 2019

1 Part III: System tests, model and state-based testing

1.1 System Testing

It is time for you to write some system/end-to-end tests. And we'll derive tests directly from the requirements document.

The requirements for JPacman are contained in `doc/scenarios.md`. They are written in an *user story* style, as suggested by agile methodologies[1]. We can still use JUnit to write system tests.

- The first system test is already written in the framework. Check the `StartupSystemTest` class that is inside of the `nl.tudelft.pacman.integration` package.

Exercise 1

- *Turn User Story 4 (suspend the game) into a working system test case. You should create a `nl.tudelft.pacman.integration.suspension` package. Add references to the scenario you are testing for each case.*

Exercise 2

- *Next, turn scenarios 2.1, 2.2, and 2.3 of User Story 2 in a working system test case in a new class. After all, having a single file with all our acceptance tests can harm the comprehensibility of the test suite. Note that you may need to use launcher's `withMapFile` method.*

  Look at the `getResourcesAsStream` method that will be used by the `MapParser` in the `parseMap` method when using the `withMapFile` map and search the Java documentation for what it does and what kind of path it expects. Supplying custom maps (i.e., smaller maps you create yourself just for testing purposes) makes this assignment much easier!

Exercise 3

- *Consider scenarios 2.4 and 2.5. Explain why it is harder to create system test cases (when compared to your previous experience with unit testing) for these scenarios.*

Exercise 4

Use the smaller map to create system tests for scenarios 2.4 and 2.5.

[1] A good book about the topic: User Stories Applied, by Mike Cohn. `https://www.mountaingoatsoftware.com/books/user-stories-applied`

To make testing easier, remember that we can test the game with a smaller *map. The default map is included in the framework, in* `src/main/resources/board.txt`*. Create a new map for testing purposes, and put it in your solution in the* `src/test/resources` *folder.*

Exercise 5

- *Answer the question in exercise 3 for User Story 3 (moving monsters).*

1.2 State Machines

In this exercise, we will experiment with model-based testing through state machine models.

Exercise 6

- *Create a state machine model for the state that is implicit in the requirements contained in* `doc/scenarios.md`*. The state chart should specify what happens when pausing, winning, losing, etc.*

  You **should** use UML notation for state machines. Consider using a UML drawing tool such as UMLet.

Exercise 7

- *Derive a transition tree from the state machine.*

Exercise 8

- *Compose a* ***state (transition) table****. Specify test cases for (state, event) pairs not contained in your diagram.*

Exercise 9

- *Write a test class for the* `Game.game` *class containing the tests you just derived from the state-transition table/tree. If you think you need additional test cases, explain why, and add them to your test class.*

  Hint 1: An easy way to create a Game class is by using the `Launcher#withMapFile`.

  Hint 2: Use (some) mocking to increase observability.

1.3 Multi-Level Games

In the last part, we are going to extend JPacman's functionality in a test-driven manner. This is your chance to try Test-Driven Development. Go for it!

The functionality to add is that after winning, up to three next levels can be played. We'll look at test cases, requirements, design, and code.

Exercise 10

- *Provide a new user story and corresponding scenarios for dealing with levels, in* `doc/scenarios.md`

  Hint: Two scenarios is enough.

Exercise 11

- *Adjust the state machine from Exercise 6 so that it accommodates the multiple level functionality. Also, derive the new transition tree.*

Exercise 12

- *Derive new test cases for this new state machine. Which test cases that you earlier designed can be reused? Which ones must be adjusted?*

Exercise 13

- *Create a new top level MultiLevelLauncher (in the `src` folder of your own solution), which is a subclass of the framework's Launcher. For now, its functionality will be exactly the same as the regular launcher.*

Exercise 14

- *Create a new `MultiLevelGame` which extends `Game`. For now, its behavior can be exactly the same as Game. Adjust the `MultiLevelLauncher` so that its `makeGame` method actually creates a `MultiLevelGame`, and its `getGame` method returns it.*

  Hint:

```
private MultiLevelGame multiGame;

@Override
public MultiLevelGame getGame() {
    return multiGame;
}
```

Exercise 15

- *Reengineer your state machine test suite in JUnit, so that it can be applied to both a regular Game/Launcher, and the new MultiLevelLauncher.*

Exercise 16

- *Now that all existing tests pass on the old and the new launcher, add the specific multi-level test cases as designed in Exercise 12.*

  Note: These tests will indeed fail, as you haven't implemented multi-level functionality yet.

Exercise 17

- *Adjust the implementation to make all multi-level tests pass.*

Exercise 18

- *Inspect and report the coverage of your multi-level implementation. If necessary, refine / improve the tests to obtain the desired level of coverage. For those parts that are not covered, explain why you decided not cover them.*

1.4 Submit Part III

Exercise 19

- *Briefly reflect on the results of your work and the JPacman framework. List three things you consider good (either in your solution or in the framework), and list three things you consider annoying or bad, and propose an alternative for them.*

Exercise 20

- *Finalize all code, inspect the warnings in GitLab, double check that all tests pass, commit, and push to git for the last time. Upload report and submit assignment. Do not forget to create a zip file to Peer containing your (anonymized) report and your final source. Double-check your zip file to make sure everything is ok!*