TI1705 - Part I

Gerlof Fokkema - 4257286
Joris Lamers - 4233042

May 7, 2014

3 Part I

3.1 End-to-End Testing

Exercise 1 *Execute the smoke test, with coverage enabled. What overall (line) coverage percentage do you get? Name 2 classes that are not well-tested, and explain why the smoke test does not cover it.*

The overall coverage percentage for the smoke test is 79% (test classes excluded). One thing to note is that the line coverage varies slightly between runs. This is because testing is done using the interactive GUI. Therefore the actions that are tested are slightly influenced by the random movements of the pacman ghosts. Two classes that are not well-tested are:

- nl.tudelft.jpacman.level.CollisionInteractionMap
- nl.tudelft.jpacman.level.DefaultPlayerInteractionMap

The reason for the limited coverage in those classes is that the GUI is only started for a limited time. These two classes are both involved in collision handling. Because JPacman is only started for a limited time no collision scenario's are encountered during the tests.

Exercise 2 *Study the acceptance scenarios for User Stories 1, 2, and 4. Turn each of them into a test case, as far as possible. To do so, use the approach adopted in LauncherSmokeTest to start the game and trigger specific behavior.*

No specific remarks.

Exercise 3 *Which functionality of story 2 was hard (or even impossible) to test? Why?*

Scenario "S2.5: Player wins" was hard to test.
We can't simply automate the test scenario for S2.5, because it would involve writing a whole algorithm for a non human player controlling JPacman.
One way we can test it however, is by customizing our Board. To achieve this, we've created the class SimpleMap. This class extends the default launcher, but it overrides the makeLevel function in order to launch JPacman with a custom board. We then shrunk the board we used to a size of 3x4 (see the map below).

```
####
#P.#
####
```

All we have to do to win now is move the player one tile to the right. After this move we can immediately test whether the game has really been won.

Exercise 4 *Now try the same for Story 3 (moving monsters). Why are these scenarios hard to test?*

One problem we encountered immediately when trying to implement Story 3 is the fact that there is no class that facilitates direct access to the NPC's present on the Board. We have also customized the board again (again, see the map below).

```
######
#G...#
####.#
####P#
```

Therefore we have created the classes SimpleGhostMap and CustomGhostFactory. Our CustomGhostFactory keeps track of all Blinkies that are created. Using it's method popBlinky() we can get a reference to the most recently created Blinky. Our SimpleGhostMap then uses this factory to create a Level based on our custom Board.
After all this preparation there was another problem we encountered: The movement of Blinky is random and cannot be easily influenced. In some cases this means we have to wait until Blinky executes a certain action before we can check our test results.

3.2 Boundary Testing

Exercise 5 *Provide a domain matrix for the desired behavior of the boundary values in the withinBorders method.*

The dimensions of the board we have used for the test are 5x5.

”x >= 0 && x <getWidth() && y >= 0 && y <getHeight()”										
Boundary			Test Cases							
Variable	Condition	Type	t1	t2	t3	t4	t5	t6	t7	t8
x	>= 0	on	0							
		off		-1						
	<getWidth()	on			5					
		off				4				
	typical	in					1	2	3	4
y	>= 0	on					0			
		off						-1		
	<getHeight()	on							5	
		off								4
	typical	in	1	2	3	4				
Result			T	F	F	T	T	F	F	T

Exercise 6 *Implement the corresponding test by using JUnit's Parameters annotation.*

No specific remarks.

3.3 Submit Part I

Exercise 7 *In your report, analyze whether the code is ready for submission: Explain check-style violations that remain (if any), provide a log of all tests passing, and include a brief assessment of the additional adequacy achieved in the jpacman-framework thanks to your new classes. Also reflect on your continuous integration server results, and your commit behavior.*

Most of our comments and code styles were already according to the standards set by check-style. Because of this our code already passed most check-style verifications. Some of our methods where a little bit long, so we had to shorten them to meet the requirement. Also we had some unused imports so it was usefull to see the checkstyles. Most other warnings were regarding magic numbers and JavaDOC missing @throws annotations.
The only check-style warnings that are left in our project are in the supplied test class LauncherSmokeTest.java and in the parameterized test, WithinBordersTest.java. The check-style warnings regarding magic numbers in the WithinBordersTest cannot be solved, since this is a parameterized test.

Exercise 8 *Create a release with appropriate version number in your pom file, git tag and push. Submit the zip and report to CPM as well.*

POM Version: 1.0
No other specific remarks