Assignment 3 - Software Quality & Testing

1.1 System Testing

Exercise 1:

See *src/test/java/nl/tudelft/jpacman/integration/suspension/UserStory4Test*

Exercise 2:

See *src/test/java/nl/tudelft/jpacman/integration/UserStory2Test*

Exercise 3:

Q: 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.
A: Testing scenario 2.4 and 2.5 is harder than the previous test cases because we have a broad map (*board.txt*) and it makes testing time-consuming. In scenario S2.4 we would need to test the same conditions four times as the number of ghosts we have (as they all have different behaviors). In scenario S2.5, the player needs to eat all the pellets and check that when he eats the last one the game ends and the player wins. Testing it with the default board would mean that we will first have to make sure that the player won't be killed by a ghost and second we will have to move it till it doesn't eat all the pellets.
Moreover, if we take a look at the `Game` class, we will notice two methods: *levelWon()* and *levelLost(),* both of them will end the game. As they both trigger the same event it is hard to test which one will be used if the game is lost and which one will be used in case the game is won by the player as they could be interchanged by mistake.

Exercise 4:

See *src/test/java/nl/tudelft/jpacman/integration/UserStory2Test*

Exercise 5:

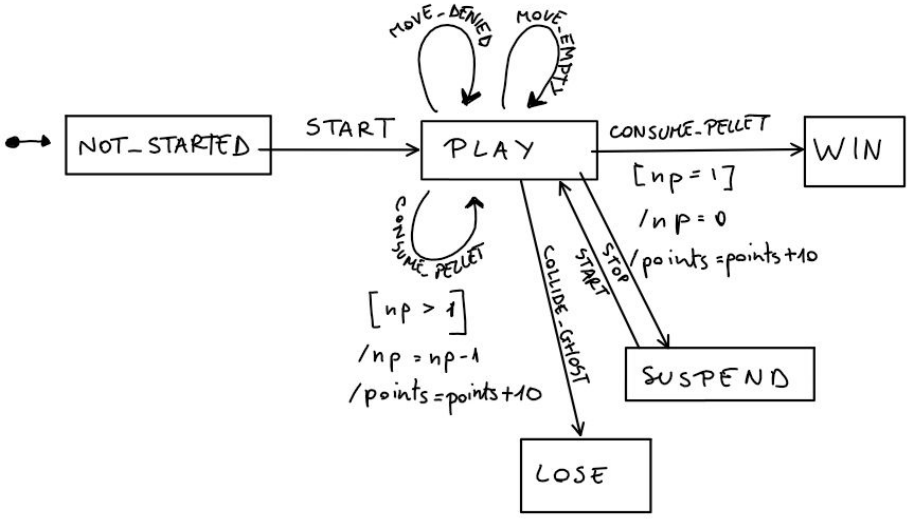
Q: Answer the question in exercise 3 for User Story 3 (moving monsters).
A: In User Story 3 we have to test the events that the ghost's moves trigger in different cases. As we are not dealing anymore with Pac-man (which is controlled by the user) it becomes harder to test each case as the workload will be multiplied by 4: as the number of our ghosts. Moreover, the ghosts move in different directions based on different conditions and that makes testing even harder.

1.2 State Machines

Exercise 6:

Q: Create a state machine model for the state that is implicit in the requirements contained in *doc/scenarios.md*. The statechart 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

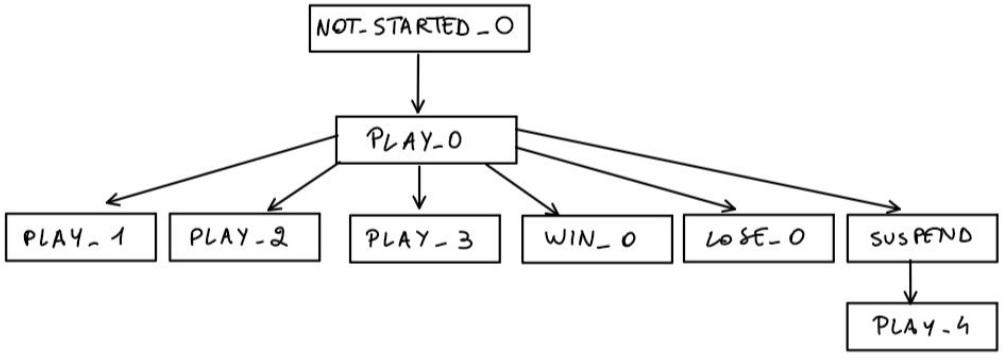
A:



Exercise 7:

Q: Derive a transition tree from the state machine.

A:



Exercise 8:

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

STATES	events						
	Start	Move-denied	Move-empty	Consume-pellet && np > 1	Consume-pellet && np == 1	Collide-ghost	Stop
NOT_ STARTED	PLAY						
PLAY		PLAY	PLAY	PLAY	WIN	LOSE	SUSPEND
SUSPEND	PLAY						
WIN							
LOSE							

Exercise 9:

Q: 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

A: See *src/test/java/nl/tudelft/jpacman/game/GameCasesTest*

1.3 Multi-Level Games

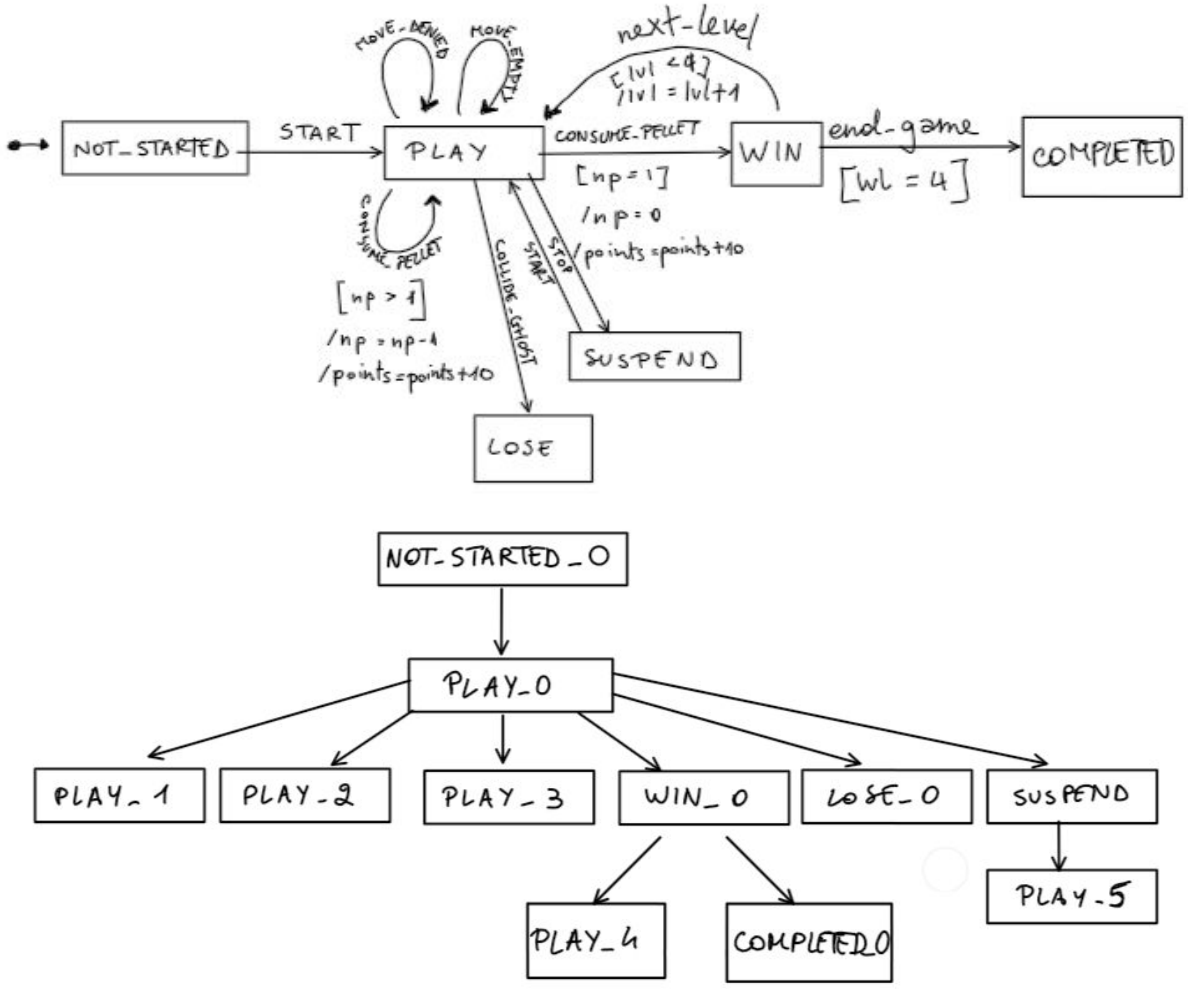
Exercise 10:

Q: Provide a new user story and corresponding scenarios for dealing with levels, in doc/scenarios.md.
A: See *doc/scenarios.md*

Exercise 11:

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

A:



Exercise 12:

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

A: Including the multi-level functionality will not change the tests a lot. The only tests that will be affected by this change are the ones that check when the player wins the game ends, this will need to be adjusted since adding new levels won't make the game end unless we reach the last one. Therefore the test currently called *testWin()* will have to be specific to the single game and another *testWin()* will have to be created for the multilevel.

Exercise 13:

See *src/main/java/nl/tudelft/jpacman/game/MultiLevelLauncher*

Exercise 14:

See *src/main/java/nl/tudelft/jpacman/MultiLevelGame*

Exercise 15:

See *src/test/java/nl/tudelft/jpacman/game*

Exercise 16:

See *src/test/java/nl/tudelft/jpacman/game/MultiLevelGameCasesTest*

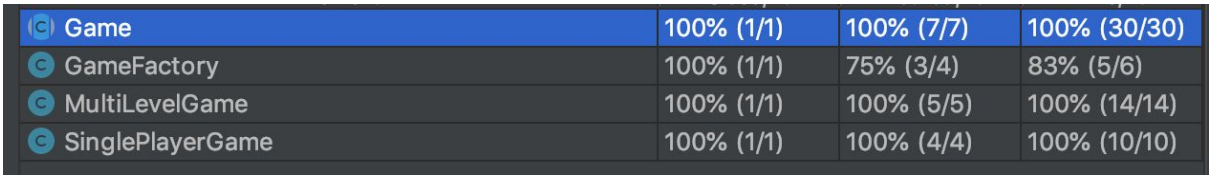
Exercise 17:

See *src/main/java/nl/tudelft/jpacman/MultiLevelLauncher* & *src/main/java/nl/tudelft/jpacman/MultiLevelGame*

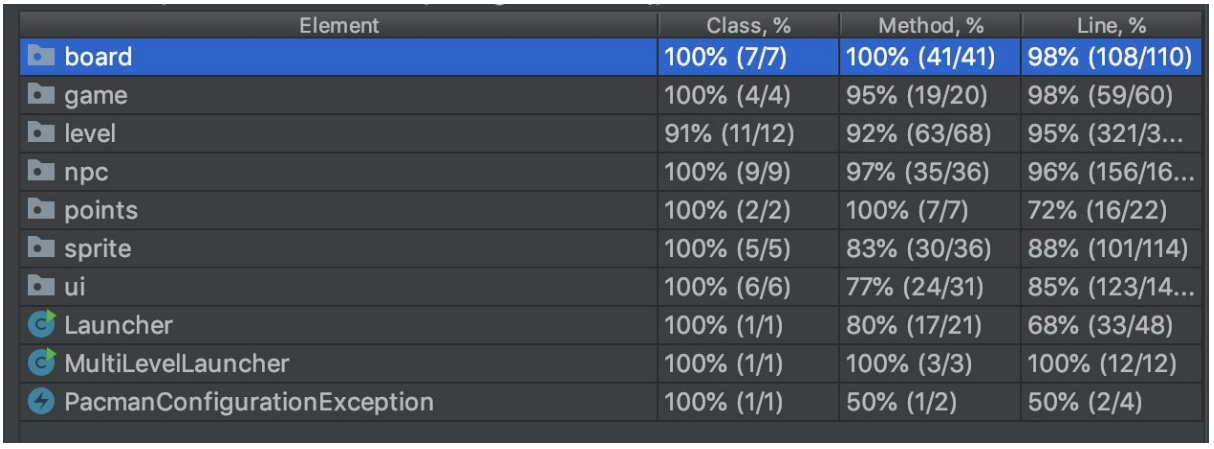
Exercise 18:

Q: 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.

A:



Game	100% (1/1)	100% (7/7)	100% (30/30)
GameFactory	100% (1/1)	75% (3/4)	83% (5/6)
MultiLevelGame	100% (1/1)	100% (5/5)	100% (14/14)
SinglePlayerGame	100% (1/1)	100% (4/4)	100% (10/10)



Element	Class, %	Method, %	Line, %
board	100% (7/7)	100% (41/41)	98% (108/110)
game	100% (4/4)	95% (19/20)	98% (59/60)
level	91% (11/12)	92% (63/68)	95% (321/3...
npc	100% (9/9)	97% (35/36)	96% (156/16...
points	100% (2/2)	100% (7/7)	72% (16/22)
sprite	100% (5/5)	83% (30/36)	88% (101/114)
ui	100% (6/6)	77% (24/31)	85% (123/14...
Launcher	100% (1/1)	80% (17/21)	68% (33/48)
MultiLevelLauncher	100% (1/1)	100% (3/3)	100% (12/12)
PacmanConfigurationException	100% (1/1)	50% (1/2)	50% (2/4)

Both classes used in the multilevel implementation (*MultiLevelLauncher* & *MultiLevelGame*) have 100% coverage which means there is nothing that still needs to be tested.

Exercise 19:

Q: 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.

A: What we consider good from the assignment is that it helps us putting in practice the knowledge we gained from the lectures so we can understand better all the different testing techniques ***(1)***. We also had the opportunity to use different tools such as SpotBugs, Gradle, PMD and Checkstyle ***(2)***. Doing the assignment in pairs helped us to improve our communication, teamwork and organizational skills ***(3)***.
What we disliked about the assignment is that some exercises are vague and could be easily misleading which could result in losing points, for this reason we suggest to write clearly what you want us to achieve in each exercise ***(1)***. Moreover, some exercises were very extensive while others were very short, maybe breaking the big ones into multiple smaller ones could make the assignment much easier to follow ***(2)***. Also the peer review and the submit site only being accessible to one member of the group caused us some problems as sometimes we couldn't meet up and we had to split the review to do it separately ***(3)***.