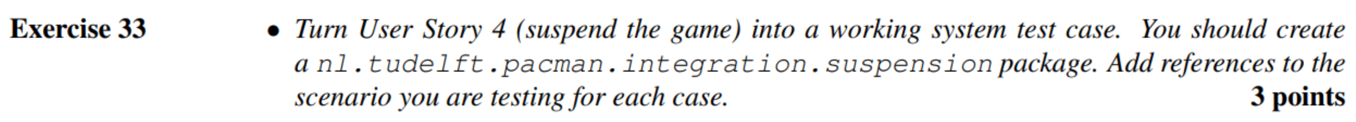
Students

- Eric Jarosch - 227271
- Lukas Fritzsche - 227771

Exercise 33



Exercise 33

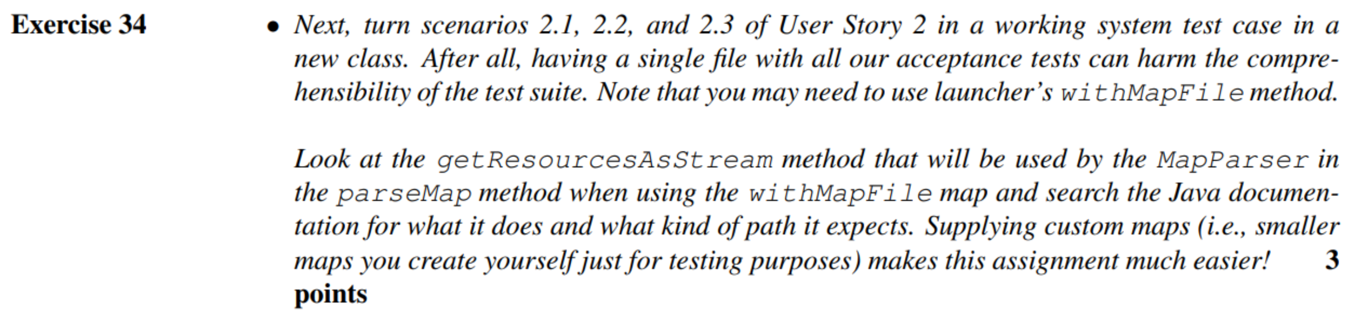
- *Turn User Story 4 (suspend the game) into a working system test case. You should create a `nl.tudelft.pacman.integration.suspension` package. Add references to the scenario you are testing for each case.* **3 points**

see added test class `SuspensionTest`
or alternatively the relevant Merge Request

Essentially, this test covers three cases:

- Initial start allows movement of units
- Suspension after initial start disallows movement of units
- Restart after suspension allows movement of units

Exercise 34



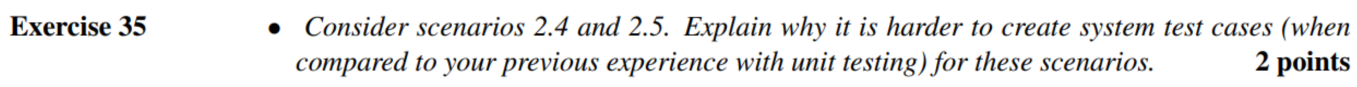
Exercise 34

- *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! **3 points**

see added test class `PlayerMovementTest`
or alternatively the relevant Merge Request

Exercise 35



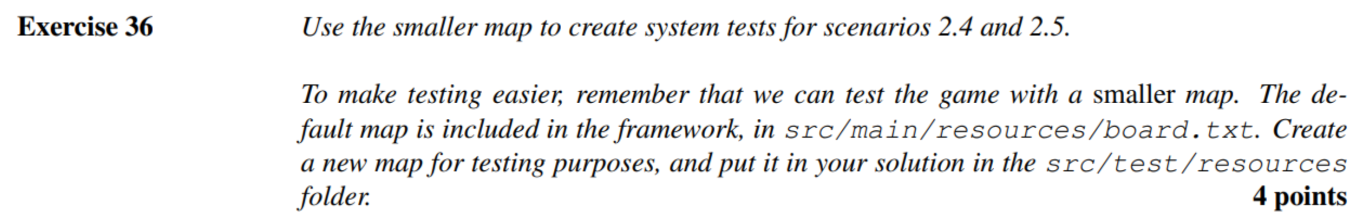
Exercise 35 • *Consider scenarios 2.4 and 2.5. Explain why it is harder to create system test cases (when compared to your previous experience with unit testing) for these scenarios.* **2 points**

Scenario 2.4 and 2.5 are mostly quite similar to Unit-Testing, since there are decently powerful interfaces to control the player. (In a way, it is also "easier" than Unit-Testing, since the System manages a large chunk of initialization for us)

However, 2.4 and 2.5 have the added difficulty that we cannot directly observe whether Player::setAlive or Level::levelLost has been called.
What we have to do instead, is to observe the attributes of the Level to identify what happened (e.g. "no pellets left" -> won | or "all players dead" -> lost)
Alternatively we can register a LevelObserver to listen to levelWon and levelLost.

2.4 provides additional difficulty, since the Ghosts might move if the thread executing the test is too slow. This could result in flaky tests.

Exercise 36

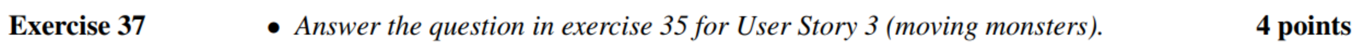


Exercise 36 *Use the smaller map to create system tests for scenarios 2.4 and 2.5.*

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.* **4 points**

see modified test class `PlayerMovementTest`
or alternatively the relevant Merge Request

Exercise 37



Exercise 37 • *Answer the question in exercise 35 for User Story 3 (moving monsters).* **4 points**

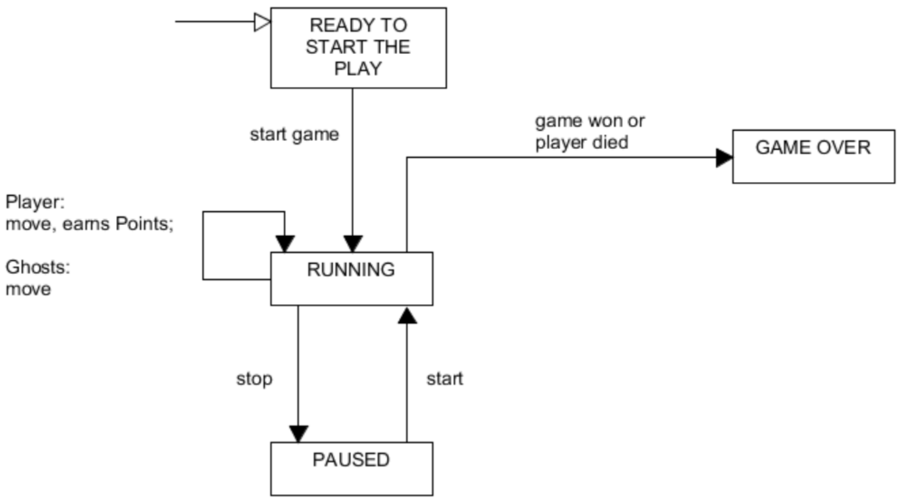
Testing User Story 3 is a lot more complicated.
We cannot directly control the action of a ghost, but we can provide custom levels in such a way that the ghost chooses our desired action.
Additionally, we have to wait for the ghost to execute its action, fortunately we can register a LevelObserver that allows us to observe when a move is made.

Overall, when performing System Tests, there are a lot more side-effects to consider than during Unit Testing.

Exercise 38

Exercise 38

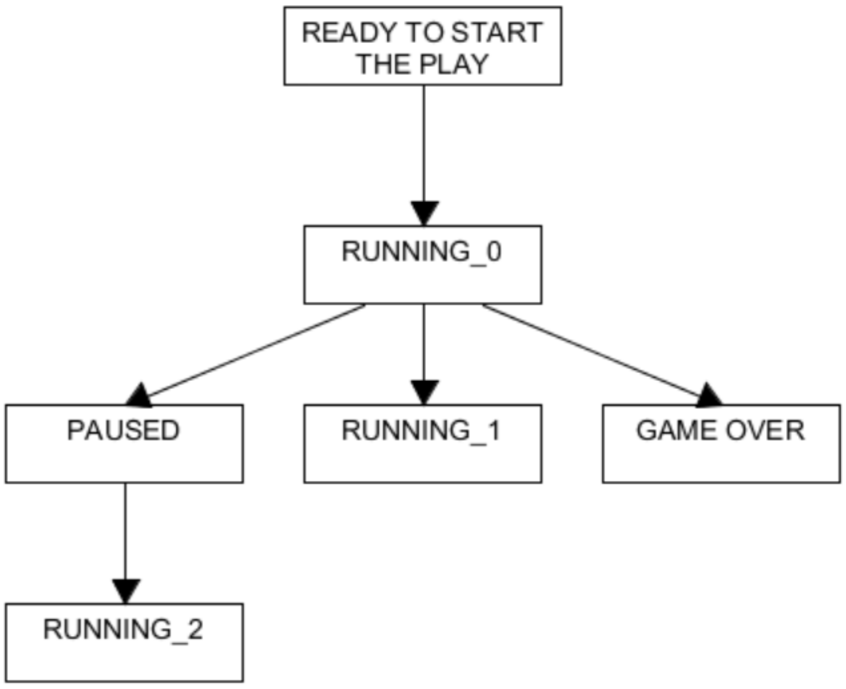
- *Create a state machine model for the state that is implicit in the requirements contained in* `doc/scenarios.md`*. The state chart should specify what happens when pausing, winning, losing, etc.* **9 points**



Exercise 39

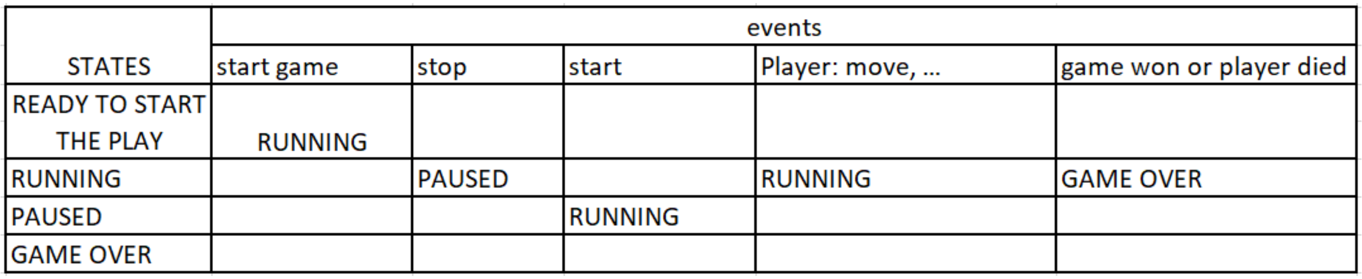
Exercise 39

- *Derive a transition tree from the state machine.* **9 points**



Exercise 40

Exercise 40 • *Compose a* ***state (transition) table****.* **5 points**



STATES	events				
	start game	stop	start	Player: move, ...	game won or player died
READY TO START THE PLAY	RUNNING				
RUNNING		PAUSED		RUNNING	GAME OVER
PAUSED			RUNNING		
GAME OVER					

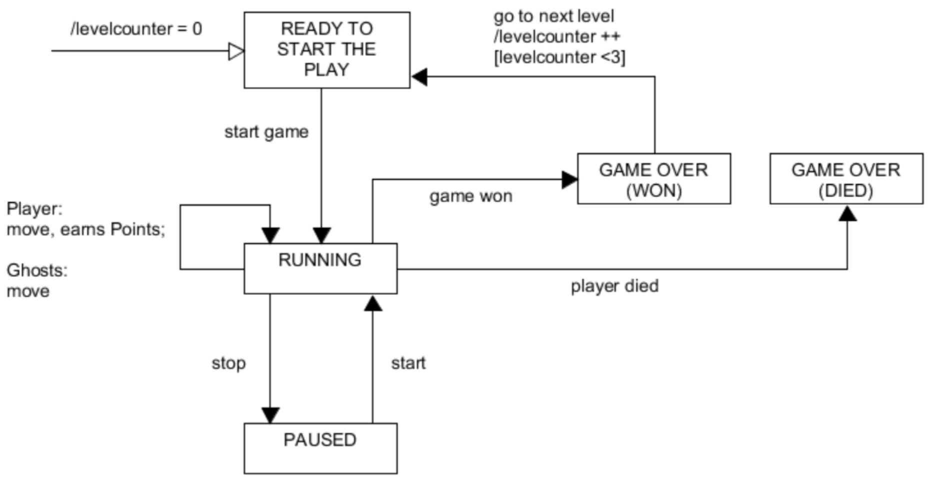
Exercise 42

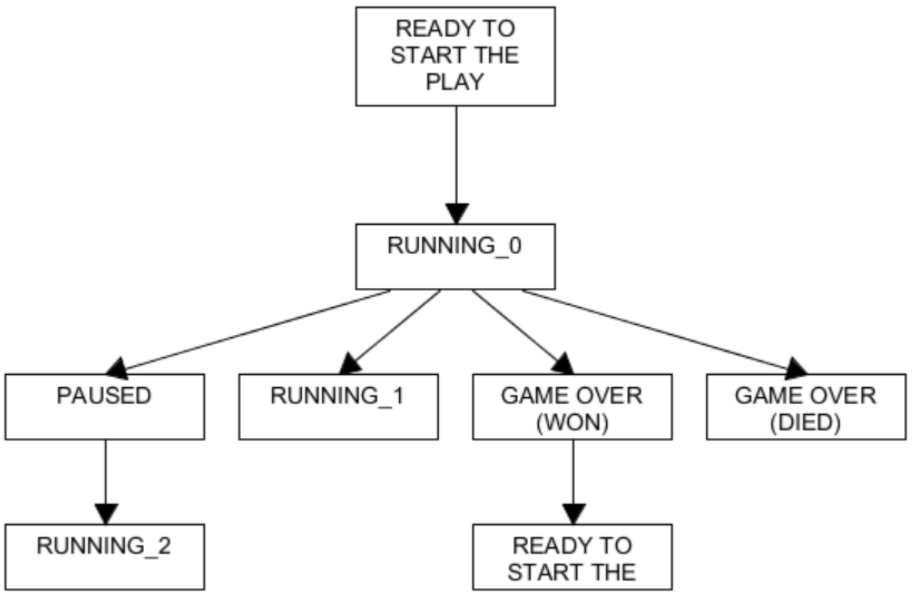
Exercise 42 • *Provide a new user story and corresponding scenarios for dealing with levels, in* `doc/scenarios.md`
Hint: Two scenarios is enough. **4 points**

see modified document `scenarios.md`

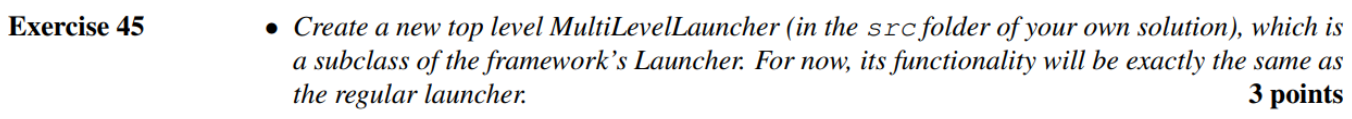
Exercise 43

Exercise 43 • *Adjust the state machine from Exercise 38 so that it accommodates the multiple level functionality. Also, derive the new transition tree.* **3 points**





Exercise 45

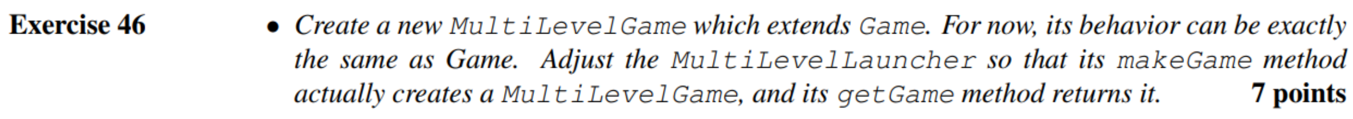


Exercise 45

- *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.* **3 points**

see added source class `MultiLevelLauncher`
or alternatively the relevant Merge Request

Exercise 46



Exercise 46

- *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.* **7 points**

see added source class `MultiLevelGame`
or alternatively the relevant Merge Request

Submission

"List three things you consider good (either in your solution or in the framework),..."

Working with the labworks was very helpful in gaining practical experience. Especially working with mocks brought me a lot.

"...and list three things you consider annoying or bad, and propose an alternative for them."

Nothing to say.