CMPT 276 Phase 3: Report

Group 14: Darrick Gunawan, Mahyar Sharafi Laleh, Hanxi Chen, Boyu Zhang

During phase 3 of our project, which takes place between March 28 and April 4, we made test cases for our game.

Overall Approach:

The approach we did for this phase was quite simple, each person would pick up a specific test, whether unit or interaction/integration, and when done, continue with another test from the backlog. This way, each person would be responsible to make sure their own tests work. Naturally, we all started with unit tests of the files we created, and moved on to the integration tests between the classes that we each built. Only once we finished with our own codes did we start branching out into testing for codes we did not work on individually, allowing us to get the bulk of the work done quickly at the start.

What We Learnt:

During the phase, we realised that some codes are harder to test than others. Firstly, the main issue with most of the code is that private variables are difficult to test. In a normal scenario, we would have had to do quite a lot of things in order to work around this, but by sheer luck, a lot of the classes we created also had getters and setters implemented as methods, even if we initially did not use any of it for the actual game to run. Having getters and setters might seem redundant at first but really helps when testing and when we want to expand our game.

Secondly, we realised that classes with higher dependencies are incredibly difficult to perform unit tests to, as there are so many factors to consider in regards to testing those classes. This means that we had to refactor a lot of our classes in order for it to be tested easier, which was luckily part of our assignment 3, which was due before this phase of the project.

Tests:

Making the tests, we tried different methods to ensure that everything in the game is tested. Firstly, we created unit tests for our classes, where we tried to do a single test file for a single class. From there, because of our code design where integrations are coupled into the classes, we were able to do some integration testing in the same files, combining unit and integrations tests in the same files, as such, the tests below will contain both unit and integration.

In the entity package, we test everything related to entities, such as players and enemies. Before testing, what each of these have in common are the way we set it up, which is spawning it in a certain area on the map. In each of these packages, both attributes and

methods are tested to ensure they work properly. For example, in both the moving enemy and player tests, we ensure that movement methods work properly, such as going in all four directions and stopping after moving. On top of that, we tested everything that was unique to each of these classes, such as player health, player taking damage, trap activation for enemy trap, and hitboxes for the enemies. This allows us to test integration between entities classes.

In the operation test package, we tested everything that the player can do in regards to the inputs. Since our game is incredibly simple, there is not much to test here. This test package checks for inputs that will be able to move the player, such as pressing the arrow keys, and releasing them to ensure that they stop.

In the sound test package, we test everything related to sound, including the music and sound effects that actions make in the game itself. The way we did this was incredibly repetitive, which involves playing the a try catch statement which will try to play the sound effect or music for a certain amount of time, and throwing an exception if the sound or music does not play. We grouped the sounds based on the classes, such as the main menu screen and the winning screen, but for some classes that do not contain alot of sound, we were able to combine them into one test class, such as both types of rewards being in one class.

In the tile package, we test everything related to the map generation and map tiles. This includes the map tiles itself, the door to win the game and the walls. In this test, we tested first if the map itself will load. Then, we tested to see if everything is drawn properly. On top of that, we tested some unique attributes to some of the items in this test, such as wall and door collision.

In the UI test package, we test everything related to the UI of the game. This test package does not cover much, as most of the other UI have been covered in other classes and this is only the leftovers. With the health bar, even though we put this under UI, as it mainly tests the UI, there are quite a few integration testing as well, as the health bar conencts with multiple classes and can only function with that connection. Firstly, we tested to see if the image itself spawns. From here, we were able to test the integration tests, which includes decreased health and death, which is no health. This test allowed us to check for both the UI of the health reacting and also the interaction between classes as well, as the health will only change if a player is damaged by an enemy.

In the menu package, we test everything related to the menu states and the running state. In all of these states, we mainly tested the button to firstly check if it is there, then check if it is clickable and doing its intended purpose, asuch as bringing us to a different game state, which can fall into integration testing. The one thing we did not test in the running state are the player inputs, as we already tested them in the operation test package.

In the item test package, we test everything related to items, which are the two different types of rewards we have. Both types of items have very similar types of testing, which tests the item values, the item being picked up, and the interaction with the player upon collision. On top

of that, the bonus test also had to be tested in regards to it disappearing and spawning again in a different location.

Test Quality and Coverage:

In order to make good quality tests, we made sure that we tested everything that is possible and reasonable for us to test, both unit and interaction wise. Furthermore, we did bounds testing to test a lot of different possible values and ensure that it works for all of the values that it is supposed to work as. This allowed us to have around 80% method coverage, an average of over 60% line coverage, which we think is fair as we have alot of imports and variable creations, and an estimate of 80% branch coverage from both our unit and integration tests.

There were some things that we did not test. This is due to multiple reasons, mainly because they are either impossible and/or unimportant. The methods that we did not cover are due to how our class hierarchy is structured, as some methods that are in a parent class gets overwritten in the subclasses, which we end up testing. This means that the method in the parent class was never actually used in the game, and we think it is fair to say that if all the subclass methods work, then the parent class method is functional.

Changes and Improvements made to production code:

Most of the changes done to the production code are minor, as they are mainly code smells from refactoring our code as part of assignment 3. After doing assignment 3, there were some smells that we missed that we end up changing for this code, such as finding more dead code during testing that we removed. There was one big change that was made in between this phase and the prior one, which was how our map generation works. While the functionality is the same, we did have to change how the map spawns in order for it to cover multiple screen types, as previously it had trouble working on smaller screens.