CMPT 276 Phase 3: Report

Group 8: Darren Jennedy, Setu Patel, Steve Lam, Samuel Wong

During phase 3 of our project, which happens between March 23 and April 15, we were tasked to test our game, “How Not to Fail University”.

What we learned:

During the testing period, we learned that specific parts of our code were difficult to test because of different access modifiers in our source code. Having private fields and methods made it difficult to test and verify various aspects of our game. Furthermore, testing abstract classes was impossible without having to instantiate them first. In our testing, we had to learn to counteract these two issues. One of the ways we managed to test the classes without changing our access modifiers was to create mock classes. This method is especially useful for abstract classes, such as our MovableEntity class. Mocking the classes we needed to test was also useful in reducing dependencies in a unit test. For example, in most of our moveable entity unit tests, we had to create a mock maze class, to decouple the tests from the actual maze class. Another method we used to counteract access modifiers was reflections. Using Java’s Class and Field classes, we managed to examine the runtime configurations of a class. This is especially useful for randomized methods which are not seeded.

We also learned that having high dependency between classes made it difficult to perform unit tests. Therefore, throughout the testing period, we refactored some classes which were highly coupled with other classes or had too many responsibilities. For instance, we realized during testing that our World class was not only generating and updating the game world, but also managed position validity. This made the class harder to test, since it had a lot of responsibilities to account for. In order to improve our tests and adhere to good design principles, we shifted some of the responsibilities of the World class to the Entity abstract class.

Unit tests:

To test functionality of a single class, we created unit tests for our classes. Our unit tests cover most of our components of the game, leaving out certain classes which deal with interaction between classes, which were tested in the integration tests, such as our GameResult classes and GameEffect classes.

In the Maze unit tests, the main features that were tested were the functions that dealt with the Point positions on the maze. Unit tests were created to verify that the Points that were generated within the Maze class were all within the bounds by using in/out-points and on/off-points.

For the collectibles, unit tests were created for each collectible class to check that the instantiation of each collectible worked, each with a corresponding position. Another unit test we had was to test that the worth of the collectibles, when collected, contained the correct value. We also created tests which ensured that setting the position of a collectible worked and that they were within the bounds of the maze.

The moveable classes that were unit tested were the Enemy, MoveableEntity, and Player class. For the Enemy unit tests, we verified that enemyValidMovement works by checking if calling that function would move the enemy to a valid position. We also tested enemyInvalidMovement by verifying that an invalid move would not change the enemy's position in order to cover all necessary conditions. Similarly for the MoveableEntity class, which covers the movement of the Enemy and Player classes, we also tested validMovement and invalidMovment for entities that move around the maze. For the Player unit tests, we checked that the getHasMoved function works where the players hasMoved status does not change until the players position is moved to a valid position on the maze.

As mentioned previously, we tested the class that was responsible for updating the game world after refactoring. We used Java's Class and Field class to reflect the runtime configuration of the world class and assert that the update functions were working as expected.

For the movement classes, we created unit tests that verifies that the Point that is generated by the Direction methods works. We checked if the generating Directions with the corresponding compass directions would accurately return the current point position relative to the maze grid system. We also tested the movement algorithm for the enemy class, which is located in the TargetedMovementGenerator. To test the class, we created a moving entity which targets a different entity, and comprehensively tested all possible movement combinations by creating different, small maps to block the movement of the moving entity and assert that the movement generated is according to the algorithm.

We also unit tested the GUI menus, asserting that each component is added correctly to the component containers. For the ending screen test, we tested the class more extensively, as it had more components than other classes, and also contained multiple nested containers, which were prone to error. First, we created a mock class for the ending screen, as it was an abstract class for both the winning and losing screens. Next, we tested not only the correct additions of the components, but also the setting of the bounds of each container, to make sure that the nested components were configured correctly.

Other classes we created unit tests for were the music classes and TimeFormatConverter. For the music classes, we verified that the audio was running when it was supposed to and also not

running when it wasn't supposed to. The unit tests in the TimeFormatConverter test if the time that is displayed correctly records the time and converts it in the right format.

Interactions and Integration Testing:

For integration testing, we focused on the most important interactions in the game logic, which consists of the entities and their positions, valid and invalid movements, game results (win, lose), game effects (point updates), and the containers which contain groups of entities.

To create the integration tests, we needed a way to control the randomness generated by the World and Maze class. We also needed a way to represent the Game class, to simulate the game effects, such as the score updates. Therefore, we created mock classes of the World and Maze class, and integrated the necessary components from the Game class to the mock World class. These mock classes have a similar functionality to their concrete class counterparts, but they introduce ways to set up the integration tests, such as modifiable maze maps and non-random entity generation.

We created five integration tests, each testing different interactions between important components in the game logic:

- ScoreTest
    - Tested the interaction between the GameEffect classes which carried the update of the scores, the collectibles which modify the score, and the player entity which has to collect the collectibles.
    - Works by setting up a small maze, filling the maze with different collectibles which yield different numbers of points, and moving the player through the collectibles.
- WinningTest
    - Tested the interaction between the GameEffect classes which communicated whether or not the player has fulfilled the winning condition of collecting all necessary rewards and the GameResult classes, which contained the information of the victory. The player entity's movement interaction to the winning condition was also tested in this test, by moving the player to the exit door before and after the exit door is unlocked.
- LosingTest
    - Conversely to the winning test, the losing test tests the interaction between the GameEffect classes, which communicated whether or not the player has fulfilled the conditions to lose the game, and the GameResult classes, which contained the information of the loss. Interactions between the losing conditions were also tested, which were the player-enemy interaction, where the player loses if it is located in the same point as the enemy, and the player-score-punishment

interaction, where the player loses after collecting a punishment which reduces the score to below zero.

- EnemyMovementTest
    - Tests the interaction between the player entity, the movement generator, and the moving enemy. The tests revolve around the enemy's interaction with the movement generator, and whether or not the movement generator causes the enemy to move towards the player. A special interaction we tested was the interaction between the player's first move and the enemy movement generator, which was to make sure that the enemy is not able to move until the player has made a move.
- PositionValidityTest
    - Tests the interaction between the position validator classes, the barrier class, and the movable entities (enemy and player). The test is to assert that the position validator works as expected, stopping the player and enemy from moving past invalid positions, such as positions blocked by barriers or walls.

Test Quality and Coverage:

In order to improve the quality of our tests, we made sure to use bounds testing whenever possible. An example of this is the maze class, where we created tests for the valid points of the maze using the bounds of the maze, testing the in/out-points and on/off-points of the maze. We also tested multiple conditions, and exhausted all possible conditions whenever possible. For instance, in our TargetedMovementGeneratorTest, we exhausted all conditions by placing walls in different directions of the moving entity to test all combinations of the movement generated by the movement algorithm.

We calculated the results of our testing to have 75% branch coverage from both our unit tests and our integration tests. We improved our test coverage by using mock classes and reflection to test classes that were impossible to test otherwise. Some components of the game were not covered by our tests, such as the Game class which contained the game loop, or the drawing methods in the World class. These components were excluded from the tests because they required player input or a running version of the game to test, which can only be tested using system and manual tests. Furthermore, not all branches of all methods can be tested. For example, we couldn't test all branches from the entity's render methods, because they required a graphical interface to render to. However, we managed to counteract this hurdle by asserting that the render did not throw an exception when run to test that the method works as expected.

Changes made to production code during testing:

As mentioned previously, a major change we made was in the World class. We changed the way we generated an empty position on the maze for collectibles by using a position validator. This change was continued in the Maze class where we generated an available Point that takes in said position validator before returning a position. These changes were made to make the classes easier to test and to spread the responsibilities handled by the classes. We also made some small changes to increase the readability of our code such as altering function names and descriptions which makes our source code easy to understand. Another change we made was to increase the enemy movement speed to increase game difficulty as we found that the previous speed was too easy for the player to maneuver around.

Bugs and Improvement:

A bug that we found was that entities spawned in front of the entrance or around it which made it possible for punishments to be generated in front of the player which would make them automatically lose the game when they walk into it. We fixed this bug by creating a class that invalidated the initial position of other entities within two cells of the player's starting position. If an entity was to be generated in the invalidated position, the position generation function would choose another position.

Another bug that we found was if an enemy moves into a barrier, they would not attempt to move around the barrier but keep moving towards the barrier, staying in place. By shifting the position validator to be an argument for the movement function instead of a fixed field in the movement generator class, we managed to alter the mechanics of the enemy movement which allowed them to move around the barriers.

There was an issue with enemies moving towards the player at the entrance before the game started which can lead to a state of the game where the enemies surrounded the player even before the player has had the chance to move. The problem was fixed by adding a new field called hasMoved for the player that is initially set to false which changes to true after the player moves. By creating this field, this allowed the enemy movement function to check the players hasMoved status until its value is true before making a move.

An improvement we made to our game was adding additional music to our game. Previously, while the game had sound effects for losing the game, there was no auditory indication for when the player had won the game. Therefore, we added graduation music that plays during the winning screen which makes it feel more rewarding when the player wins the game. Adding these sound effects and background music helps create immersion within our game.