Implementing Test Cases

Our group started off our work by organizing all classes and their capabilities into a document so we were clear and organized on what needed to be tested. This list included classes like GamePanel, MapGenerator and Player; and capabilities like Player Mechanics, Goblin Mechanics and MapGeneration amongst others. Once we had this list, we made unit tests that made sure all behaviours worked as intended and integration tests that made sure interacting classes worked as intended.

To organize the test classes, we ensured the segregation of test cases for each package which made it easier for each of us to work on separate packages for unit tests and helped our group keep the test classes as organized as possible. For example, tests involving the Player class were written in testPlayer class. Features that integrated multiple classes were tested in whichever class was more involved in the action.

We spent a lot of time learning how to master the use of Mockito, once we figured out how, it allowed us to simulate mock events. The use of mock events solved a lot of problems with our tests as it allowed us to not rely on the implementation of the method and create a mock scenario that we wanted to test directly instead. We were able to test individual methods directly instead of relying on their interaction with a different class. For example, simulate key inputs to test player movement without relying on direct keyboard inputs. This method was used in ObjectManagerTest as well, in this case, mock events were used to create mock objects which were used to test functions with, for example, testing removeObject() by creating and removing a mock object. Using Mockito made many processes easier and more concise.

Many of our classes contained void methods and private attributes which made writing tests directly impossible, so to test these components, we implemented helper subclasses that

extend the parent to let us override some methods such as setGame() to test their functionality. This strategy helped us since every method we wanted to test in a class could be overridden by a subclass and tested as needed making our tests concise and straightforward.

One of the biggest challenges was testing our graphics components, paintComponent requires a valid graphics context and it is only valid when something is rendered on the screen. This made it challenging to test. We eventually tested paintComponent by using a buffered image, testImage, and used it to create a mock graphic object which was passed as the parameter for paintComponent. A similar solution was used in tests involving the draw() function which is present in all visible components of the game.

Tests focused on the Goblin class were mainly focused on pathfinding, collisions and animation changes. To help with this we temporarily implemented a “debug mode” which could be accessed by pressing ‘f’ while playing the game, using debug mode would visually represent the radars, distance to goblins and the expected path of the goblin; being able to visually perceive these elements were crucial in making problem-solving easier.

Tests involving the Player class focused on player movement, collisions and animation changes. Mockito was used here to simulate key inputs to test all the elements involved with the class.

In tests like CollisionCheckerTest, we added a function moveEntityToPosition, that let us force interactions between game objects that we wanted to test, for example, a collision between the player and goblin could be tested by moving the goblin to the player position.

At the end of our testing, we reached **96%** Line Coverage and **92%** Branch Coverage

Package Name	Line Coverage (%)	Branch Coverage (%)
App	95	88
Entity	98	98
Keyboard	98	94
Objects	90	100
PathFinder	95	86
Tile	96	93
UI	95	90

Overall, we are satisfied with our test cases, while making these tests we realized that our code was organized as expected, we spent a lot of time towards the end of phase 2 ensuring that our project followed software engineering guidelines; this amount of time spent on organizing ensured writing test methods was straightforward and we could write test cases that reached expected coverage.

Along with the JUnit automated tests, we performed many informal play tests to ensure various elements in the gameplay were performing as intended. This process made us intimately familiar with the elements, processes and flaws in our game.

Bug Fixes

While running our unit and integration tests, we discovered a few bugs that helped us identify mistakes. The biggest of the bugs involved collisions, we found that when collisions occurred between the Player and Goblin from certain directions, it was not registered. This was a

very important discovery since the collision between the Player and the Goblin was a crucial part of the game, maybe the most important action of the game.

Changes to our Code

We did not feel the need to make any major changes in our code because the overall organization of our code was mostly done during phase 2, the changes that were made in phase 3 involved changes that improved code quality by removing redundant methods. Upon running our tests and examining our coverage, we concluded that the steps taken in phase 2 to make sure we followed software engineering principles paid off; we believe that our code was adequately organized.