Test Automation

Important individual features of the game:

- **Player**

  The user controls the player to collect rewards. After all the necessary rewards are collected, the user navigates the player to the exit door on the board. The player can move right, left, up, and down. These movements are tested in the **PlayerTest** class. This also includes the different set and get methods for both the grid values and pixel values.

- **MovingEnemy**

  The moving enemy is one of the significant units of the game that the user would interact with. The moving enemy chases the player throughout the game. Like the player, the moving enemy can move right, left, up, and down. The movement of the moving enemy is handled by the updateEnemy method which is discussed later in this report. All of these are tested in the **MovingEnemyTest** class.

- **BonusReward**

  The BonusReward inside the game boosts the players score by 500 points. They appear randomly inside the game at random intervals and disappear after a few ticks. The tests were based upon the basic functionality and overviewed different scenarios of the implementation. Tests were also implemented to check if the bonus reward has been collected or not. Some other tests focussed on checking whether the bonus reward occupies the same place as their chosen x and y coordinates. The final feature that was tested was the static BonusReward list methods which update a list that holds multiple BonusRewards. All of these are tested in the **BonusRewardTest** class.

- **Reward**

  The regular rewards get displayed on the Game Screen once the game begins. With the value set as 100, the player has to collect all the rewards to win the game. The tests associated with the reward follow the same trend to check situations of reward collection and whether the location of the reward is the same as chosen coordinates. These tests are done in the **RewardTest** class.

- **Scoreboard**

  The Scoreboard keeps track of the timing for menus as well as the overall score of the user. Singleton design was used to implement the scoreboard.The tests checked to see if methods involving time were behaving as designed. Besides that, we have implemented tests to check whether the score held by the scoreboard is as anticipated. The feature is tested in the **ScoreboardTest** class.

- **Stationary enemies**

  The stationary enemies are located in special locations on the board. If the player occupies the same tile as a stationary enemy , the player incurs a loss of 300 points.. The test checks the expected behaviour by first placing the player and the stationary enemy on the same cell, and then placing the player and the stationary enemy on different cells. The feature is tested in the **StationaryEnemyTest** class.

- **Tile**

  Tile is how we can check the values each coordinate on our grid holds. This was done using an int to keep a number which represented a type and by having the tile know it's own position. The other thing we tested was if one tile's location equals to the given tile. These tests are done in the **TileTest** class. The tile was also tested heavily in TheGrid class as it is a main component of that class.

- **TheGrid**

  The grid holds a 2-D array of tiles. The tests were based on how we can update the grid using Tile and how we update other variables in it. These tests would run to make sure the TheGrid was updating properly with the different calls. We had to make multiple get and set methods for the tiles because our grid 2-D array held everything in simple int values but Slick2D wanted exact pixel values to draw onto the screen. Another major test for TheGrid was collisions which was a bunch of different integration tests for the class. Tests for TheGrid were stored in **TheGridTest** class. Like tile, TheGrid was also tested heavily in other classes as it needs to be constantly updated.

- **Tick**

  Tick controls the overall flow of the game. Each update of the game is directly dependent on the tick cycle. One tick takes 1/32 of a second and the game state updates every 4 ticks(⅛ of a second). Therefore, it was crucial to test if the Tick class behaves in synchrony with the computer's inner clock. The Tick class works on a separate thread invoked by TickThread. The test was done by putting the thread to sleep and checking if the passed time increments the tick counter by 1. The feature is tested in the **TickTest** class.

Important interactions between features:

1) Moving enemy and Player

Throughout the game the moving enemy chases down the main character. If both player and moving enemy happen to be on the same cell location, then the player loses the game and the failure menu appears. The moving enemy is immune to other enemies or rewards on the board. Like the player, the moving enemy cannot pass through inner or outer walls. The interaction between these two classes is tested in the **IntegrationMovingPlayerTest** class.

2) Collisions detection

The collision detections required 6 different methods, 1 for each tile type collision and an extra one for the player and enemy colliding. This was done by creating new objects for each tile type to test if, one the player interacts with it and, two it has the correct interaction. The reward, bonus reward and stationary enemy all used the score board class too as we needed a way to check if the score updated correctly. These tests were done in **TheGridTest** class.

Measure we took to ensure the quality of the test

To ensure the quality of the tests, every test was aimed for a 100% line and branch coverage. Proper analysis of the tests looking at different scenarios were tested and then the tests with low coverage were analyzed. If a person faced failures for the test, the tests were later reviewed during the group meeting to ensure that the test passed with complete coverage.Besides that, the quality of the test was improved by testing only the important and working features of the game.

Analysis of testing

We were able to get 100% line coverage for most classes, and 100% method coverage for all of the single features and interaction between those features. We ensured that all of the methods and logic worked properly as expected. The classes we tested were ones that had public methods that were not tied down to the GUI, so not every class got a JUnit test class.

Class	Class Coverage	Method Coverage	Line Coverage
BonusReward	100% (1/1)	100% (18/18)	100% (50/50)
MovingEnemy	100% (1/1)	100% (11/11)	94% (69/73)
Player	100% (1/1)	100% (8/8)	100% (21/21)
Reward	100% (1/1)	100% (11/11)	100% (22/22)
Scoreboard	100% (1/1)	100% (9/9)	100% (26/26)
StationaryEnemy	100% (1/1)	100% (11/11)	100% (20/20)
TheGrid	100% (1/1)	100% (15/15)	80% (86/107)
Tick	100% (1/1)	100% (7/7)	100% (14/14)
TickThread	100% (1/1)	100% (2/2)	93% (14/15)
Tile	100% (1/1)	100% (7/7)	100% (16/16)

Features or code segments that are not covered

All the classes related to menus since it was not part of our implementation and was dependent on the GUI. The only way we could test these menus was to do manual testing. Same problems we had with the draw methods. The drawing of images depends on the external library and we were able to test these methods only manually.

Findings

The main criterion for designing test cases was to aim for a 100% line and branch coverage. We tried to exercise every single unit of functionality in our project. Soon we realized that it would not be possible. The reason being is that a significant portion of the logic was tied in the private methods of the GameScreen class. That did not allow us to test the main interactions and units of the game. To test our game rigorously, we decided to refactor some of the private methods outside of the GameScreen.

One thing we found out early on about testing was for the GUI. We had no way to test out our GUI methods in GameScreen. This caused us to refactor the GameScreen so we could move methods out of it and into a class that made it easily testable with JUnit.

The updateEnemy method was refactored into the MovingEnemyClass. The decision led to a better design for future extensions of the class to create special types of enemies with unique movements on the board. Along with this we also refactored our stationary, bonus reward and reward lists as we decided they were too important to be stuck in the GameScreen.

Collision handling was another method that was refactored out of the GameScreen and this it was moved to TheGrid class. We thought this decision would help our game be more modular and make it easier on development of extra levels. The original function had no drawing, we were able to move it by making the underlying 2-D array list static so it can be accessed by all classes. This also caused us to move some booleans out of the GameScreen and into TheGrid so they could be changed in the function.

Array lists that kept track of multiple instances of the stationary enemies, rewards, and bonus rewards were moved outside of the private variables of GameScreen into their respectable classes. The decision led us to be able to test if the arrays have proper instances of objects and increase the overall line coverage.

During manual testing we found bugs with how the bonus rewards were spawning. They were appearing on top of other types of tiles because of how their location is always random. We found the bug after a while and deduced it down to not a mathematical error but an error on the order in which we were placing down tiles.