# Group-12 Phase 3 Report

Authors: Wilson, Daniel, Nicholas, Yuxi

## <u>Unit and Integration Tests</u>

During our testing phase, we attempted to test as many methods as we could. We first planned to test every single class and most of the methods by using JUnit Tests. Once the JUnit tests were finished, we then worked together to create integration tests for our code. The work was divided as the same as Phase 2.

Within the GameKeyListener class, we tested the various keyboard inputs the user has access to, and ensured they all responded correctly. The method getLastKey assisted with this as it saved the last key pressed from the KeyPressed Method. ResetLastKey was also tested because it allows the user to choose to make no input by the end of the timer.

Since the state changing depended on the keyboard inputs of ENTER and ESCAPE, we decided to create an integration test. We wanted to ensure that changing from the GAME state to the MENU state was working. Also, pausing the game by returning to the MENU paused the game successfully. Furthermore, a getState method was added to our GameMain Class to help with the testing process.

We tested the main character extensively because there are many boundaries to its health and status. The initial health of the main character is set to 100. We reduced its health using its decHealth, testing when decHealth was called greater than or equals 100  times, and less than 100 times, ensuring the main character produced the correct health. Furthermore, we used the method checkAlive which returns a boolean indicating if the character is alive or dead. The test class also has an integration test by using the getter and setter function created through the BoardEntity class because the main character itself is a board entity (MainCharacter extends BoardEntity).

The BoardEntity class handles all the coordinates of the collectibles, enemies and main character. We created a new instance of the board entity, and checked that its default position X and Y position should first be at (0,0). Using the parameterized constructor, we could also set the coordinates of an entity at any location on the board. We also ensured the getter and setter methods were working as intended as well to set the X and Y positions of an entity.

The Board class is in charge of creating the board, and creating the cells within the board. The Board class also handles the inBounds method of our game which ensures no entities can leave the board or move onto a BARRIER cell type. We first created a new board, and tested whether inBounds returned the correct boolean result. An integration test was used to test the  Cell types of BARRIER and OPEN. The cell types are created through the Cell class. We used the various getter and setter functions to test that the cell sizes, cell types and cell positions were all correct.

The Cell class creates the cell types and contains the getType and setType methods for the cells. We tested the types and ensured the right type was returned.

In CollectibleTest, because it inherited from BoardEntity, we carried out some tests to confirm that it succeeded in inheriting some of the functions in BoardEntity, including getXPos and getYPos. In addition to location information, different collectibles also contain different properties. We test the default values of these properties and the correctness of the settings to ensure that they can correctly perform their duties in the game.

For Enemy Class, because it is relatively simple, we only tested its default value (similar to Collectible) and found that it did not inherit properties in BoardEntity when testing. So we modified it to inherit the property of boardentity and have the same function.

Testing for the TickTimer class was challenging since it is meant to run as a separate thread and most of the methods were private. The testing that we did for it was fairly basic, simply testing the getters, the tick reset method, and the pause/unpause tick method.

GameMain testing was also difficult, due to its messy interwoven methods and status as a Singleton design pattern. Testing was broken into two lopsided methods, with one unit test simply to ensure the Singleton property was met, and the other an integration test executing most public methods in the order they are required - extra mostly-private initialization from startGame and the thread-necessitating timer-based update method being especially important.

## Test Quality and Coverage

To ensure consistency in our tests, all group members followed a template containing imports and naming conventions. We tried to cover as much of our code as possible with the tests with an end result of 80% line coverage. Some methods in the GameMain class were private and/or not realistically accessible resulting in a big chunk of the missing coverage. In order to reach a high branch coverage we tried to have assertions for all possible cases where it was reasonable. We achieved a branch coverage of 60% (130 branches out of 216). We tried to cover branches that made the most sense and ignored the ones that didn't make sense to test.

## Challenges

We wanted to cover as much test code as possible. However, many functions are placed in GameMain Class, and they are done in private , and the JUnit Test can only test methods which are public. We can only try to test some functions in series, so as to test their wholeness. But there are still some functions that we can not test, but we have passed most of the code tests, and found some problems and bugs through the tests and made modifications.

## Findings

Within the GameKeyListener class, we found out that uppercase and lowercase characters have different key-codes. This stumped us during our testing phase since the game ran without these issues at first. We later learned that the keycodes depended on whether KeyPressed or KeyTyped was used. In our testing cases, we covered both uppercase letter and lowercase letters.

We did improve our code production by adding getters methods to missing setters methods, and also removed a useless setter method.

A large bug that we discovered in our testing was a circular method call in our TickTimer class. Our game tick was filling the stack with function calls that never had a chance to finish execution because a new function call would happen before the original could end. We fixed this by changing TickTimer to a while(true) structure.

In Enemy class, it inherits the properties of BoardEntity. Through testing, we find that it can not invoke the functions existing in BoardEntity, which let us find this bug and modify it.

In the process of testing Collectible, we found that the data stored in Punishment is damage, and we affect the score in the game, so we modified it.