Game logic unit and integration tests

All subclasses of the ‘Character’ class, such as ‘Boss’ or ‘MainCharacter’ had many simple methods that could be unit tested, like methods to increase their health or attack, or to take damage and reduce their health. They also had methods that interacted with other characters, thus the need to create a specific character for them to interact with. ‘MainCharacter’ would need to attack an object of some subclass of ‘Enemy’, and any ‘Enemy’ subclass would need to attack an object of ‘MainCharacter’.

All subclasses of ‘Item’ were also simple to unit test. Their main method would be to activate when an object of type ‘MainCharacter’, and act accordingly depending on that object’s state. Some examples would be how ‘RegenHeart’ only activated and disappeared when the activating ‘MainCharacter’ was not on full health, how the ‘SpikeTrap’ activates regardless of the ‘MainCharacter’s health, or how ‘Key’ only activates when it is enabled and obtainable. One exception that is grouped in with this section is the test for the ‘Backpack’ class. ‘Backpack’ is the data structure that is contained in the ‘MainCharacter’ class and stores what items that character has. Thus the test class for ‘Backpack’ only tests it’s ability to add and remove items to its inventory.

For testing the UI elements, like ‘MoveCountdown’ and ‘StatIncreaseIndicator’, these were also straightforward to unit test.These tests mostly consist of ensuring that specific sprites are enabled and disabled at the correct times.

Next is the game’s logic section. First, the tests for ‘MenuScreen’ and ‘Map’ are somewhat simple unit tests. The tests for ‘MenuScreen’ are similar to the tests for the UI elements in that the point of the test is to call a method, then verify that the correct sprites are enabled or disabled. The tests for ‘Map’ include setting and getting tiles in the map, and clearing the whole map.

The ‘GameLogicDriver’ class is what stitches all the parts of the game together and runs the game. Naturally, it will have many methods that it uses which needs to be tested, and each of those methods may have multiple different outcomes depending on the game’s situation. Almost all the tests use other objects like characters or items, and so are somewhat dependent on them. The ‘GameLogicDriver’ also stores the game’s map, all the game’s characters and items, etc. so most of the tests involve checking if there are any of the specified objects in a specified position, or removing an object from the game.

Game logic test coverage

For subclasses of ‘Character’, with the exception of getters and setters and overridden methods related to rendering, the branch coverage is about 90% and the line coverage is about 95%. Getters and setters are trivial, and it is hard to verify that methods related to rendering did their expected behavior. There are some branches/lines that are not covered only because they

are executed when undefined behavior happens and a print statement is executed to notify of that behavior. The test for the method “canEnemyMove” in the ‘Enemy’ class is contained in the test class for ‘GameLogicDriver’ as testing that method required accessing protected fields in ‘GameLogicDriver’ which could not be accessed in the test methods for the enemies.

For subclasses of ‘Item’, the branch and line coverage for all classes is 100% with the exception of setters and getters. Similar to before, the test for the “activate” method in the ‘Exit’ class is contained in the test class for ‘GameLogicDriver’. This is because calling ‘Exit’s activate method when the activating character has a key results in ending the game, which requires running protected methods and checking protected attributes of ‘GameLogicDriver’.

For UI elements, the branch and line coverage is also 100% for all classes. These classes were easy to cover all lines and branches as there aren’t very many of them.

In terms of the game’s logic, all methods in the ‘Map’ class related to setting and getting tiles are 100% tested. The methods related to pathfinding were not covered during this phase because they were tested separately during the pathfinding test.

For the ‘MenuScreen’ test class, the line coverage is about 85% and branch coverage about 60% because there is one method that involves the user pressing space. This method cannot be tested by itself. Lastly for the ‘GameLogicDriver’ test class, the line coverage is about 80% and the branch coverage is about 75%. Again, most of the branches and lines that aren’t being tested are ones that only result in extraordinary circumstances and would not be possible due to checks from other parts of the code. There are also some code segments that cannot be tested independently due to limitations by the game’s engine resulting in errors.

Game logic findings

After all of the testing for the game’s logic, not much was needed to be fixed. One code quality fix that was made was abstracting a shared attribute between all the objects that display sprites on the menu to a single abstract class. Another was splitting up a large method into smaller function calls in the ‘GameLogicDriver’ class to make it more readable.

Pathfinding Tests

When testing the pathfinding algorithm, significant changes were made to optimize the algorithm. These significant changes were executed in two phases.

Phase1:

- Unit and integration tests:
    - During this phase, the whole algorithm was built in the Pathfinder class with a single method that returns the next position that an enemy is supposed to go to when trying to reach the main character.
    - This made it easy to unit test:
    - The major test cases done during the unit test: were checking if the output of a function is not null, an enemy does not step on a wall tile, or an enemy doesn't get stuck between tiles wall without a way to get out.
- Test coverage:
    - The test coverage was 95%. The algorithm was made of hard-coded scenarios, which led to easy testing of all the scenarios.
- Test findings
    - Though the test coverage was significant, the results were 50% correct. This made us notice and conclude that the pathfinding was not working correctly.
    - Most of the major functionality was not functional. For instance, an enemy could get to the player's exact position. The enemies could get stuck between walls. The pathfinder was not optimal.

This situation motivated the change of the whole algorithm, and the approach used to solve it was adequately implementing the A* algorithm for pathfinding instead of hardcoding a solution.

Phase2:

- Unit and integration test:

    - The Pathfinder class, which is made of a simple method, was unit tested during this phase.

    - The test cases covered were similar to those in phase 1, with a much higher success result. Taste cases such as the enemy trying to step in the same tile position as the main character worked. The enemies were following the character properly.

- Test coverage:

    - The major parts of the test coverage was mainly done on the findpath method. This method was related to other helper functions in the map class. By testing the findpath method, those functions were covered. Most of the bugs that were revealed by testing were fixed.

- Test findings:

    - During testing, we stumbled upon errors of infinite loops, but testing made sure we corrected them.