Maze Game Project - IT'S RAW!
CMPT 276 - Fall2023

Group 24 Phase 3 Report

Rashed Hadi
Timmy Tsai
Raymond Chan
Raymond Zhou
Due: November 29, 2023
School of Computing Science
Simon Fraser University

Unit and Integration Tests

Our focus for the third phase was testing the game and fixing bugs. We first identified all the features that needed to be unit tested and ran automated tests.

- **ScoreTest.java:** ScoreTest focuses on testing the scoring mechanism related to the interactions between the main character and specific objects (rewards, bonus rewards, punishments) in the game. A setup method is created to create an instance of a game panel and a main character. In each test method, regular rewards are tested by creating a reward object, and placing it in a specific location and in an obj array. The main character pickup method is called on that object and assertEquals is used to ensure the score is properly adjusted. For regular and bonus rewards, the score count increases by 1 and 2 respectively, while punishments result in a decrement by 2.

- **RewardSetterTest.java:** RewardSetterTest tests the placement of rewards in the game, specifically the positions of different reward types. An instance of the game panel is created and calls the necessary methods to set up the board. assertEquals is used on each object type that already exists inside the array for the main game panel and tests if their positions are the expected outcome.

- **OxygenLevelTest.java:** OxygenLevelTest tests if the oxygen level system behaves properly. This class contains methods that test if oxygen levels decrease over time or increase when picking up oxygen tanks and if the game state changes to the lose state if oxygen reaches to 0. Before each test method, we have a setup method so instances of the game panel and the main character are created. Each test method uses assertEquals to ensure the expected oxygen level is the outcome.

- **FileSystemTest.java:** FileSystemTest contains a series of test methods that check if the correct image and sprites are loaded successfully when new objects such as rewards. Punishments, or main characters are created. The class uses assertNotNull and will output “x image not loaded successfully” if the image is not properly read.

- **MainCharacterTest.java:** The testing for the main character consists of 7 tests that start off by testing the main character’s default positioning on the board, his movement in all 4

directions, his ability to collide with and pick up rewards, and finally his oxygen level over time. Together these tests cover all of the main character's fundamental actions that he performs in our game. Test #6, that tests his ability to pick up rewards, indirectly allows us to test the main character's collision with all objects such as the enemy, end-game portal, and harmful objects.

- **EnemyTest.java:** The enemy in our game is a character of few methods since the shark's logic is mostly based in the A Star search algorithm that is tested separately. There were only 2 tests that tested the other aspects of the enemies functionality, the first tests the set_default_position method which in fact ended up doing a lot more than solely setting the enemies default position. It takes care of setting up all the default variables that the enemy needs to function which are tested using assertEquals statements. The second test ensures that the shark's sprites were loaded properly into the game by using assertNotNull statements.

- **AStarTest.java:** Our A Star search algorithm that the enemy in the game uses to find the player while they are moving around the screen is an essential part of our game. We tested the algorithm using 2 integration tests and 1 unit test. The unit test ensures that our cellCost method, an essential part of the algorithm that calculates the cost of the path to that cell, is working properly by comparing the calculations it produces to a manually calculated example. On the other hand, the two integration tests test the rest of the methods that are called within each other while the algorithm searches for the path. The first integration test, provides the path finding algorithm with a regular scenario with a target cell and an initial cell and ensures the algorithm finds a path between them. The second test provides the algorithm with a target cell and an initial cell, then proceeds by blocking all the cells in the board, making it impossible for the algorithm to find a path, ensuring the algorithm is able to consider situations where there is no possible way to get to the target cell.

- **SuperObjectTest.java:** SuperObjectTest contains multiple test cases that assess different aspects of the "SuperObject" class. We test the drawing functionality by ensuring the draw method does not throw exceptions, checking the default state of the collision attribute, verifying the collision attribute after modification, assessing the default values of the solid area, and checking the default world coordinates. Every test case

uses assertions from JUnit testing, such as assertNotNull, assertTrue, assertFalse, and assertEquals.

- **CellTest.java:** The cell class in our game is responsible for defining the necessary attributes of each tile that makes up the game board. To create the cells (tiles), this class assigns unique identification numbers (ID) to each type of texture, a texture image, and a collision property (boolean can be true or false). In total, we have 3 unique cell textures. This class contains one constructor and one getter for the cell texture ID. Given that we have 3 unique cell textures, the constructor was written with switch case statements for easy selection of the desired texture when calling the constructor. To thoroughly test the methods, each unique texture has its own unit test where it tests the creation of the object by comparing the expected texture ID, its texture image by comparing RGB values of each pixel, and its collision status whether it is true or false. Please note that for our textures, the sand has no collision (boolean set to false) and both the rock and seaweed textures are set to have collision (boolean set to true). As switch case statements were used for the constructor, a test for invalid texture selections was also implemented to exhaust the default case of the switch case.

- **CellCollectionTest.java:** With our CellCollection class, it is responsible for holding the various unique textures that are required to be imported into the game. To hold the various textures, it contains an array of type Cell in order to order and store each texture for later use. This class only contains one constructor method that calls for the construction of the Cell class to store references of each texture type. A unit test case was made to test this function as well as the integration of this class with the Cell class. More specifically, it verifies the order of which the textures are added. The order of the cell textures are determined by their texture ID.

- **BoardTest.java:** BoardTest.java proctors the generation behaviour of the Board class of the game. By generation, the board class is responsible for using the imported cell textures to draw and map out the whole game board. It holds a 2D array that stores a unique cell texture ID to distinguish the type of cell that is situated at a given (x,y) coordinate. To test this class, it was necessary to modify the generation functions in the Board class so that the cells do not get painted in the game window and also eliminate the need to use any Graphics or Graphics2D objects. In total, there are two tests. Firstly,

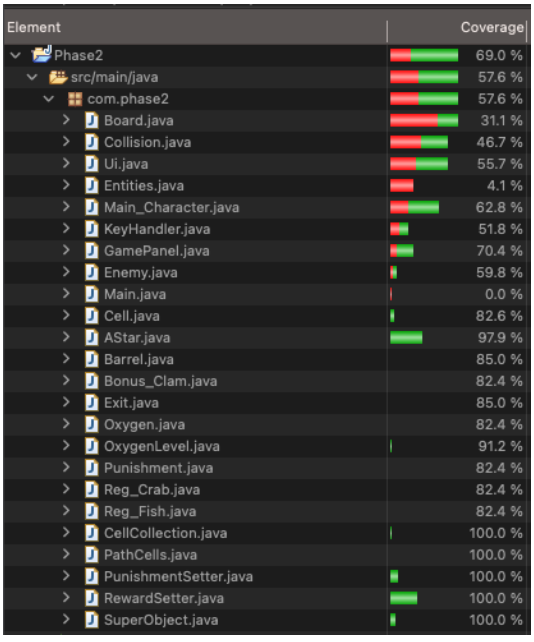
the generation of the floor and perimeter of the board was called and tested. Moreover, the next test calls for obstacle tiles to generate onto the plain game board. For both cases, the cell textures are individually added into the game layout array in the Board class when generating the layout. The tests compare the generated board layout arrays with the expected layout arrays on every index of the arrays.

- **KeyHandlerTest.java:** KeyHandlerTest has a series of tests to check if Keys in the game are performing properly from the “KeyHandler” class. There are a total of 9 tests including KeyPressed, KeyReleased, titletest, gamePause, gameContinue, and gameRestart. These tests cover all the keys that you can control during a game. AssertTrue and AssertEqual are used to check the tests. Moreover, the test cases had a lot of repeated code, so refactoring was done to simplify the code. We have created helper methods such as setup, PressKey, and ReleaseKey to improve test quality.

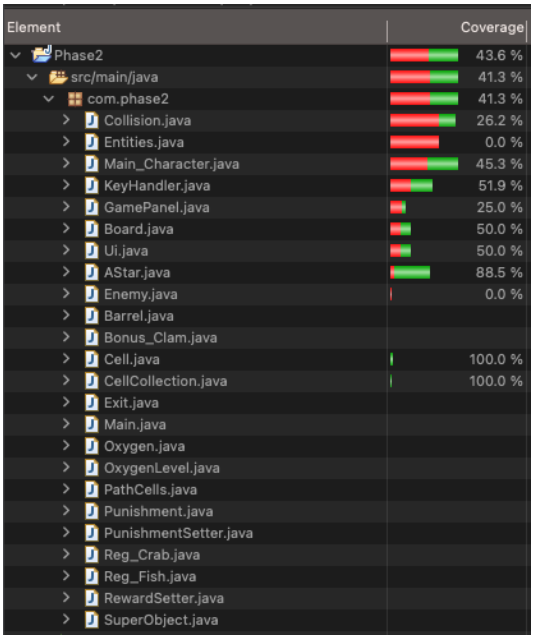
Test Quality and Coverage

To ensure high test quality of the code, we used various refactoring in our code and test cases to improve the efficiency. We did not use automated testing for testing the user interface because we thought it would be more practical to test UI by running the game and using random functional testing such as different inputs to observe the outcome. Our total tests had 59.5% line coverage, 41.3% branch coverage, and 57.6 instruction coverage.

Line Coverage



Branch Coverage



Findings

- We found that we needed to change the way our file system worked. Originally for each file read we used “image = ImageIO.read(getClass().getResourceAsStream(“path”);”. However, after writing our test classes, we noticed that we were getting null input errors after running our tests despite having the correct file paths. After changing our file read to “image = ImageIO.read(new File("path")); and changing our path, we fixed the issue.