Nov 26th, 2022

CMPT 276 Project Phase 3
Testing: Report

Maze to the Queen Bee

Group 16

Marco Lanfranchi – 301433830
Sana Dallalzadeh Atoufi – 301399601
Satvik Garg – 301452798
Jinshuo Zhang – 301385111

Contents

Testing Report

Managing the Testing Phase

For the testing phase of the project, all group members worked on creating tests for the classes and methods that they were responsible for designing in phase 2. We were not limited to testing these classes though; we worked together to help each other create tests as well to so that we could achieve higher test coverage. In the next paragraphs, all 4 group members will describe the work they did for the testing phase.

Jinshuo's Report

Unit and Integration Tests

For Phase 3, I did test for Enemy class. The purpose of Enemy class is to place some enemies on the map randomly (not including Bee's start point). Then all enemies will start to chase down the Bee (Player) until the Bee finishes the game. So, I decide to make two parts on my Enemy test. First part is to test if the enemy can be placed on map randomly. Second part is to test if the enemy can find the right path and move to the target location. First part is easy, testEnemyIsPlacedOnMap() function is to get some random numbers as (X, Y) location for placing the Enemy. Both X and Y should greater than 0 and smaller than 23*tileSize.
Second part is challenging. A* path-finding algorithm works by dividing the map into multiple pieces, for each piece is a node, then calculate the distance between the node's location and enemy's location also the distance between the node's location and the target's location for each node. Then find the shortest path for enemy, then the enemy will start moving. Therefore, I start by using testNodeIsInstantiated() to initialize nodes, then by using testNodeIsSetted() to test if node is able to setup. After these two test, function testGetCostFunction() is to get two distance which are for node to contain. For now, nodes contain all distance that can be used to find a path. Then is the part for testing if the enemy will move successfully. But before test the movement, collision needs to be test first. I use testEnemyCanMoveUp/Down/Left/Right() and testEnemyCanNotMoveUp/Down/Left/Right() in total 8 functions to test if there is collision happens when the enemy is moving Up/Down/Left/Right. Then testEnemyMovesUp/Down/Left/Right() in total 12 functions to test if the enemy will find the correct direction in different situation. Since the enemy's speed is not equal to one tileSize when it is moving, then there will be some situation that the enemy will get stuck by a wall when it is passing to other rooms. Therefore 12 functions will cover all situations that may happen during chasing down the Bee(Player).

Test Quality and Coverage

In Enemy class, there is a function called searchPath(). Which contains the enemy's moving part, but it is unable to test all different situation because of the random variables inside that function. I have tested this function by same values three times, and the result is that I have got three different moving situations in searchPath() function. This function is like a set of

different steps of A* pathfinding, therefore from placing the node, until reaches the goal location, it can not be broken and manipulated inside this function. And also function enemyCheckCollision() has been called under searchPath() function for some special moving situation, which can avoid the enemy from stopping by the wall while it is moving to another room. For the rest function of Enemy Class, they are full covered and tested.

Sana's Report

Rewards test: This test file had to do with the rewards generator, which uses assertions to test whether all the rewards for the characters are generated properly on the map. The rewards will make our main character collect points resulting in a higher overall score. The main aspect of testing was whether the correct number of rewards would be generated and if their locations would be properly placed (not all on top of each other). We also tested for the correct number of regular and bonus rewards being created and checked for position overlaps. In our game there are many important interactions between the classes as all the classes are reliant on other classes or will need to work with other classes in some way.

Measure taken to ensure the quality of our test cases: As mentioned, all of our group members had a fairly good understanding of the things that they wanted to test for since the 2nd phase of the project when we assigned classes to different group members. Therefore, we employed certain measures while writing test files to ensure high quality test files. We tested all aspects of certain classes not leaving anything untested. Even though this phase was less eventful than the previous one we still learned a lot about testing.

Satvik's Report

Unit and Integration Tests

For Phase 3 of this project, I was in charge of writing tests for the classes I created in Phase 2, which included the Environment/Lighting classes, Tile Manager, Map Creator, various pick-up object classes, and some other classes that I worked on with my group members, such as GamePanel, Keyhandler, and Bonus reward. I began by writing unit tests for the TileManager and Environment classes. The TileManager class was in charge of producing the map's basic text layout using 1's and 0's. So my first tests were to see if the TileManager's constructor was operating properly, since it instantiated the mapTileNum variable, which housed the text file in an array format. I also tested other TileManager functions, such as loadMap, which was responsible for getting the file from the specified path and iterating through filling up the mapTileNum variable. For this, I wrote a couple of tests: one to ensure that the mapTileNum variable was not null when loadmap was called, another to ensure that each value within the mapTileNum variable was not null, and finally to compare the mapTileNum values to the text file to ensure that everything was properly assigned. These were the tests testSetMap, testMapTileNum, and testMapAssignNumbers respectively. These types of unit tests, in which I tested each function separately, were also performed for classes such as Environment/Lighting and Map Creator.

The tests I'd like to highlight are for the draw methods. My group and I were first constrained by the fact that you can't test UI because there's nothing to assert and that you'd need a Graphics2D object to draw the UI on. However, after digging through the Graphics2D object and the internet, Marco and I discovered a technique to build a temporary image and then make a graphics object from it. We were able to test our drawing method without having to launch the game. For example, in the TileManagerTest class, the testDrawMethod function was an example of an Integration test because I merged the TileManager class's setMap and draw methods. I made a temporary image and then a Graphics2D variable out of it, trying the draw and setMap methods. This template was then applied to all of the other draw methods in the other classes. For example, in the GamePanelTest class, I tested the drawReward and drawObject methods in the same manner as I did in the UITest class.

Test Quality and Coverage

I used several of the techniques we learnt in class to ensure that the tests I developed were of high quality and covered as much of the code as feasible. To begin, I used assert statements throughout my tests to ensure that any differences or errors in my code were caught. One of the tests I wrote in the Keyhandler class, for example, examined if the game states updated properly on key presses. So, for the test, I'd set up the variables and key presses to trigger various parts of the code, and then use the assertEquals method to determine if the gamestate was correct.This prompted the use of a different system, the domain testing technique, which we studied in class. In this, I examined the various outcomes that a function or condition for a loop could provide. For instance, to enhance game performance, the Enemy draw method only drew enemies that the user could see from their present position. So, in order to test the result, I divided the condition into 4 conditions and tested each one separately in order to determine the result and raise the coverage score.

I was able to get 100% Line coverage for my Environment and Lighting classes, 99% Line and 96% Branch coverage in the MapCreator class, 98% Line and 81% Branch coverage in the TileManager class, as well as increase coverage in many other draw methods across the different classes. Many of the losses in coverage percentage were caused by try/catch block and conditions that couldn't be untrue, which prevented the branch from operating and reduced coverage percentage. However, I think that by including more test cases across the system, I was able to raise the total coverage score.

Marco’s Report

Unit and Integration Tests

For Phase 3 of the project, I was responsible for testing the classes and methods that I designed in Phase 2, being the Entity class, Bee class, KeyHandler class, and some more classes that I worked on with my partners. I began my testing by creating a test class, EntityTest, for

testing the Entity Class. The features in Entity are shared by Bee and Enemy, so I knew right away that in making tests for this class, I would need to test features that concern an entity's movement and location around the map. This test class makes use of an Entity with a specific map which is initialized before each test is executed. I started by making simple unit tests, testSetX() and testSetY() to assert that the methods for setting an Entity's x and y points were setting the correct values. I then made unit tests to test a method that returns the tile number of the tile on which the Entity is located. These test's name's start with testGetTileNum. In each of these tests, the Entity's location is set on a different type of tile on the map (location determined simply by looking at the .txt map used for this test class) and I then asserted that calling Entity's getTileNum() method returns the correct number according to the type of tile. The next set of unit tests for this test class are similar to the previous ones except they were for testing a feature in Entity that returns the tile number of the tiles to the left, right, up, and down directions of the entity. Entities are designed to move freely, with the exception of going through walls. I tested this feature by making 8 integration tests in the EntityTest class whose name's start with testCheckWallCollision. Individually, these tests assert that when any 4 of the directions around an Entity are a wall (or not), their boolean moveDir value is set to false (or true). The last set of tests I made in EntityTest were unit tests for a method in Entity that returns true if the entity is located in the first room of the map or not.

The next test class I made, BeeTest, was for testing the Bee class and its methods. This test class makes use of a Bee with a specific map which is initialized before each test is executed. The first set of tests I designed in this class, whose names start with testUpdateSprite, are used to assert that the Bee's sprite number is correct according to another field in Bee, the sprite counter. This feature is used to switch the bee's image back and forth to illustrate flapping wings. The next set of tests I designed in the BeeTest class were integration tests testing Bee's update() method while the bee is on top of a reward or not. Then came testing one of the game's main features: moving the Bee. Bee's moveBee() method checks: if a key is pressed, and if the bee can move in that direction, before then moving the bee or not. I designed integration tests for every possible combination of the previously mentioned 2 conditions since these are all of the different possible ways a player will try to move the bee. Next I designed a set of 4 unit tests whose names start with testCheckReachedEnd and are made to test Bee's checkReachEnd method which outputs true if the bee is located on the final tile, located in the bottom right corner of the map, or false otherwise. Following this, I designed a set of integration tests whose names start with testCheckGameWon and who test Bee's checkGameWon method in many different scenarios. Many of the next tests I designed in BeeTest were to test the interactions between a Bee and the Rewards in the game. Test methods in BeeTest with names starting with testPickupReward are integration tests that I designed to test Bee's pickUpReward method. Bee's pickUpReward method makes a call to another method, OnReward, which returns true if the Bee is located on a Reward and false if not. Therefore the next set of tests I made were unit tests with names starting with testOnReward. There were multiple cases that I tested in these unit tests because a Bee's position while on a Reward can be any of the 4 corners of that Reward's tile. Another method that is called in Bee's pickUpReward is hasAllRegRewards which checks if the bee has collected all of the regular rewards, so the next set of unit tests I made were to test this method by calling it when the Bee has 0, some, and all regular rewards and asserting the boolean value it outputs. A feature that I made in Bee and tested in BeeTest was the method nearQueenMissingRewards which returns True if the Bee is within 3 tile-lengths of the Queen Bee (destination tile) but does

not have all of the regular rewards, so that a message can be displayed reminding the player to collect all rewards. The next set of tests I made, with names starting with testCheckPunishmentCollision, were to test Bee's checkPunishmentCollision method which is called inside Bee's update() to check if a Bee is on top of a punishment tile, and if so calls another method to reduce the Bee's score. These integration tests assert that the Bee's score changes when on a punishment tile, and remains the same when the bee is not on a punishment tile. I then designed tests testReduceScoreValid and Invalid to test Bee's reduceScore method in cases where the score should and should not be reduced. The final set of tests I made in BeeTest whose names start with testDraw where to test Bee's draw method for all 8 of a Bee's different images.

I made the KeyHandlerTest class and worked on it with Satvik to test the KeyHandler class and all of the possible Key inputs for all of the different game states of our game. For the title screen state, control screen state, pause state, game state, win state, and lose state all possible keyboard inputs were tested individually. We made tests to assert KeyHandler sets the correct values when keys are both pressed and released by the game player.

Test Quality and Coverage

To ensure that the tests I designed were of good quality, I used many of the testing tools mentioned in class. Firstly, each test I designed made use of at least one JUnit Assertion to assert that the code I am executing was yielding expected outputs. The testing strategy that I used the most throughout my test classes was the simplified domain testing strategy. For example, in the EntityTest class, when writing tests for Entity's inFirstRoom method, I needed two considered an upper and lower boundary for the entity's x location, and an upper and lower boundary for the entity's y location to indicate if it is in the first room of the map. I split these boundaries and tested them each separately. Another example of when I used the simplified domain testing strategy was in BeeTest, specifically for the testCheckReachedEnd test methods, where there was only one boundary condition for x and one boundary condition for y. I also used this testing strategy in BeeTest for testCheckGameWon, where the two variables are the location and number of rewards collected and I was able to break the tests into 4 separate cases. I used the simplified domain strategy more in BeeTest for the testPickupReward test methods and testOnReward test methods. In BeeTest, I also used the Modified Condition/ Decision Coverage to identify the important test cases while making the testNearQueenMissingRewards test methods. I did this because the nearQueenMissingRewards method can be tested in 8 different cases, but I wanted to obtain the 4 test cases that cover all paths of the method.

Because of these testing strategies used, I was able to obtain 100% line and branch coverage for the Entity class with its EntityTest class, and I also obtained 99% line coverage and 98% branch coverage for the Bee class with its BeeTest class. The missing line coverage in Bee is from an exception that is not handled in Bee's setImages method. The missing branch coverage for Bee comes from two simple else if statements which are redundant but are kept in the code for reader understandability.

Findings

While writing and running my tests, I did not make very many changes to the production code other than the removal of many unused fields and methods and the changing of the accessibility of some fields. I did learn however that there are some simple and some not-so-simple aspects of the game that could use redesign. For example, almost every class has an X and Y field if that class represents something with a location and in testing, I was setting X and Y values extensively. We could have therefore used a Point class to hold those values and represent a location to simplify the code. From running the tests and working on test automation, I learned how to use Maven to run all tests as well as JaCoCo to generate code coverage reports on the tests.

Note: All the test coverage percentages mentioned throughout this report are based on the JaCoCo test coverage reports that we continuously generated to check how far we were in testing.