Phase 3 Report - Group 21

The project used JUnit and Maven for testing, and both unit tests and integration tests were written. The main objectives of the testing phase were to ensure the quality and functionality of the game, identify and fix bugs, and improve the overall code quality. This report focuses on the quality of the tests and code coverage and presents the findings obtained from the testing phase.

Test Quality

To ensure the quality of the tests, we followed several measures. First, we defined clear objectives and scope for the testing phase to ensure that all aspects of the game were covered. We also automated the testing process using JUnit and Maven to reduce the chances of human error and increase the speed of the testing phase.

Furthermore, we used a combination of unit tests and integration tests to test the different components of the game. Unit tests were used to test individual functions and methods, while integration tests were used to test the interactions between different components of the game. This helped in ensuring that the game was functioning as intended and that all components were working together seamlessly. We also wrote clear and concise assertions to ensure that the test results were easily interpretable and understandable.

Finally, we assessed the code coverage of our tests by looking at the line and block coverage. As we covered more interactions, we ended up increasing our line and block coverage. It served as a good reference for how much of our code was being tested.

Code Coverage

While writing our tests, we mainly focused on the interactions rather than a metric such as line/block coverage. We felt that this was the most beneficial way to test whether our game was functioning correctly. But we did also utilize the line and block coverage metrics to assess the quality of our tests. The more interactions we covered, the more line and block coverage we were able to achieve. Below is the overview of our code coverage across the different classes of our project.

KeyHandler.java: The coverage measured was 83% (200/242). The segment that was not covered was System.Exit() and the method called, KeyReleased.

System.exit() wasn't tested. KeyReleased function does not have to be tested since the test methods in the test file declare the key press by integers and once it is used, it automatically releases.

AssetSetter.java: The coverage measured was 100% (373/373)

Items.java: The coverage measured was 100% (21/21)

Tile.java: The coverage measured was 100% (6/6)

Sound.java: The coverage measured was 97% (86/89)

PathFinder.java: The coverage measured was 99% (513/516)

Punishment.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

Coffee.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

BubbleTea.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

Stopwatch.java: The coverage measured was 71.8% (94/131). In stop() method, we only tested when the timer was running and then stopped so, inside of if/else was not tested.

Characters.java: The coverage measured was 77.5% (169/218)

Portal.java: The coverage measured was 39.5% (64/162). draw() method was not tested since it only draws on the screen.

Student.java: The coverage measured was 76% (404/531). draw() method was not tested since it only draws on the screen.

TileManager.java: The coverage measured was 55.2% (181/328). draw() method was not tested since it only draws on the screen.

GamePanel.java: The coverage measured was 40.9% (138/337). The lines of code that are related to Threads were not tested. Also, the method that is related to draw() is not tested.

Raccoon.java: The coverage measured was 59.4% (319/537)

CollisionChecker.java: The coverage measured was 44.5% (461/1035)

UI.java: The coverage measured was 6.9% (50/728)

When it comes to adding unit tests, it can be challenging to do so for classes that are more graphics-related. This is because graphics-related classes often have a visual output that is not easily testable with traditional unit testing methods which is why some classes were left out of unit-testing.

Findings

1. We have learned that finding the proper, efficient and effective tests are not easy. Finding the interaction between classes and methods was quite complicated since coupling.
2. Almost everything is connected to the GamePanel class and refactoring this would involve a huge rewrite.
3. During Phase 3, our codes were not necessary to change our code. Could not find any bugs in the class files.
4. However, as mentioned in Finding 1, due to the high coupling style code, it was not easy to create test suites for all the classes and methods. This could be the reason why some of our coverage is not high enough.

Breakdown of our Important Tests

Test file name: ***StudentTest.java***

Interaction between KeyHandler and Student Movement

Whenever input is made by the player, the keyhandler is updated to reflect the key which has been pressed. As a result the direction variable in the

Student class is also updated. Therefore, to ensure that the direction of the student is properly updated, it is important to test whether pressing the arrow keys updates the direction correctly.

Test method name: *testSetUpMovment()*

Interaction between Student and Game States

Whenever the student's health goes to 0 or the score becomes negative, the game must end. Therefore, to ensure that this logic is correctly implemented within the game, the interaction between the student and game states must be checked.

Test method name: *testUpdate() -> in StudentTest*

Interaction between Student and Items

Whenever the student collides with an item, the score and health must be updated accordingly. Therefore, to ensure that these variables are properly updated, it is important to test the functions related to picking up the items.

Test method names: *testPickUpRewards(), testPickUpPunishments*

Test file name: *RaccoonTest.java*

Interaction between Raccoon and Student

Whenever the raccoon comes in contact with the student, the collision of the raccoon must turn on. This is important as this will affect whether the player takes damage and the game ends. Therefore, it is important to check the interaction between the raccoon and student where they are at the same position.

Test method name: *testCheckCollision()*

Interaction between Raccoon and Pathfinder

Whenever the game is running, the raccoon must chase the student. For this to work, the interaction between the raccoon, student, and pathfinding class are all important. Therefore, the interaction between these components is tested to see whether the raccoon takes the shortest path towards the student.

Test method names: *testSetAction(), testSearchPath*

Test file name: *KeyHandlerTest.java*

Interaction between Keyboard input and UI design -- Title Page

Whenever the game has launched, the tile page shows up first then the user chooses the next step by pressing the keyboard (up, down, left, right). The title page (UI design) interacts with the keyboard input. There are two options

which are “NEW GAME” and “QUIT” to choose from the user. By pressing the up arrow key and the down arrow key, the user can choose either one. Therefore, it is important to test whether pressing those two keys properly navigates on the title page.

Test method name: *titlePageInteractionTest()*

Interaction between Keyboard input and UI design -- In-Game

While in the game, the user can press either ‘p’ or ESC from the keyboard to pause the game. This has to be tested to determine whether those two keys are properly working to pause the game. Furthermore, when the user presses those two keys one more time, the game has to resume.

Test method name: *inputPauseTest()*

Interaction between Keyboard input and UI design -- Game-Over

After the user dies or accomplishes the tasks in the game, the screen changes to the Game Over UI screen. On that screen, the user has to choose either “RETRY” or “QUIT” by pressing the keyboard (up and down). It has to be tested to determine whether pressing those two keys properly navigates on the options on the Game Over page.

Test method name: *inputGameOverTest()*

Test file name: *TileManagerTest.java*

Interaction between Tile - TileManager - text file

TileManger reads the **text file** that has the map information and then they are stored as a 2d array to put the tiles onto the In-Game screen. It is important to test because the map information text file is essential since it has all the information about the map. Therefore, without proper implementation of file interaction, the map would not be properly shown.

Test method name: *mapArrayTest()*

Interaction between Tile - TileManager - image files

Once the above test passes, the TileManager loops the map Array and sets the proper image files onto the tile array (image file array). It is important because proper image files that correspond to their specific number have to be loaded into a tiles array to properly represent the map on the In-Game screen.

Test method name: *imagesTest()*

Test file name: CoffeeTest.java, BubbleTeaTest.java, PortalTest.java, PunishmentTest.java, CharactersTest.java

Interaction between class - image files

The class, Coffee, BubbleTea, Portal, Punishment and Characters, get their own corresponding images. It is important to test to determine whether their images are properly set.

Test method names: *imageTest()* → in CoffeeTest, BubbleTeaTest, PunishmentTest

testGetImage() → CharactersTest

imagesNotNullTest() → PortalTest