CMPT276 Phase 3 Report

Sean Chan - smc26
Kaj Grant-Mathiasen - kgrantma
Ashhal Vellani - ava47
Bavneet Hothi - bkh6

1. Unit Test Features

For the monster class, the main area that required unit testing involved the creation of the different types of monsters and their default values as well as the main update function. As this instantiation was independent of other areas, we could therefore test this independently via unit tests. As for the update function, we tested each individual case of movement (left, right, up, down) as well as the change in sprite images. This tested the public facing update() function, as well as the internal logic involved with calculating new movements. We created these unit tests within the MonsterTest.java file, specifically the defaultCheck(), and updateCheck() functions.

For the player class, the main area was testing elements such as their default values, gold, etc. These methods were independent of the other features, so as a result we were able to test individually.

Next, we also created unit tests for the CollisionInstance class. While this technically requires functionality from the Entity and Object classes, we still classify it as unit tests. This is because it uses extremely basic logic and the tests created specifically for this class are simply for verifying we are extracting the data required for logic in other areas rather than the logic and interactions itself. These tests are contained within the CollisionInstanceTest.java file.

The Tile package was exercised using two test classes, TileTest and TileManagerTest. Tile includes a series of tests that assert subclasses of Tile generated from Tile.makeTile() are correct instances of the various subclasses, a test which expects a NullPointerException when attempting to make a tile of null type, and a test which exercises a single branch of the Tile.draw() method. Tile consists of 11 tests

In order to ensure that the game was functioning properly in different states, we employed the KeyHandlerTest class to test all keyboard inputs required to trigger various actions. The KeyHandlerTest class simulated different key inputs and then checked whether the expected game state changes occurred correctly, using assertions. For instance, when the ESC key was pressed in the Play State, the game state should transition to the Pause State and vice versa. Since the keyboard inputs were independent of other game areas, we were able to conduct unit tests and isolate individual screens and states, making the testing process more efficient.

In addition to verifying whether the game states were changing correctly, we also tested the GUI to ensure that UI components that belonged to a particular screen were drawn and updated correctly every time the game state changed. This was done using the UITest class.

For the objects, most of them were tested individually by testing whether they were instantiated correctly. The 2 classes that were not tested individually were the superobject and diamond class.

We also tested sound which did not rely on any other classes. These were mainly checks to ensure that there was correct instantiation and whether the behaviour was correct.

The asset setter had unit tests for ensuring that the getters were working correctly and that the correct object was being instantiated.

For the gameconsole class, we created tests to ensure that the gameconsole was being instantiated correctly and that the gametime was correct.

Another class that was tested independently was the utility class. This class only scaled images so the only test was to ensure that the image was scaled correctly to a 48x48 image.

2. Integration Tests

The monster class had several integration tests as there were multiple interactions between different systems within our project. We created a test that verified the drawing method and the interactions between the core game functionality such as gameConsole and Player.

The player class had several integration tests. These were player creation, updates, and draw methods that required the instantiation of the player class. Since the player class needed a keyhandler and the gameconsole, these were not possible to test individually.

The diamond and superobject classes relied on the gameconsole, so they were integration tests. The tests were instantiation and update checks to see if the diamond was correctly hidden or not.

We also created tests for gamesetter. This created multiple objects so we ensured that the correct objects were instantiated.

The tests in gameconsole were mainly to ensure that the behaviour was correct. We created methods for setup, restart, and updates.

We also created integration tests for the collision class. This is because there is a significant portion of the functionality within this class dedicated to the interaction between the player and monsters/objects/walls. Tests were created for a variety of scenarios involving the detection of collision and the picking up of objects. We tested cases where the player detects walls in each direction to verify both the getPossibleMoves() method as well as the detectWallCollision() method. Within the testPlayerGold() method, we verified that placing an object to just past each of the 4 directions of the player and the player direction set to the respective direction enabled the pickup condition collision detection. These tests are located within the CollisionCheckerTest.java file.

3. Code quality of test cases

To ensure quality of our test case code, we took several steps to mitigate duplicated code, maximize functional coverage and ensure that our test cases were comprehensive, accurate, and effective.

First, we began by planning our test cases carefully. This involved identifying the different scenarios and actions that could occur in the game and developing a testing strategy that covered all possible scenarios. We also ensured that our test data was valid and relevant to the different game states and actions, using realistic and diverse game inputs to simulate different scenarios and game states.

To ensure comprehensive testing, we made sure that our test cases covered all possible scenarios, including edge cases and error scenarios. We recognized that these scenarios could be the most challenging to detect, so we deliberately developed test cases that stressed the code and evaluated its behavior under extreme conditions.

4. Code coverage

In terms of coverage on the TileTest, we chose not to get 100% coverage. For each of the subclasses of Tile, we chose not to test the catch blocks as they were simple printStackTrace() calls. Three of the branches in the draw method were not exercised as it was deemed unnecessary to exercise them. makeTile()'s default case was not exercised as passing in a null value causes a NullPointerException, which is currently expected behavior.

We were able to get line coverage to 98% for the Entity class as we covered all but an exception case which is invoked if the image asset is not found. Branch coverage for this class reached the 100% goal.

For coverage in the MonsterTest, we were able to get line coverage to 100% and branch coverage to 89%. For the missing branches, they all involved a direction or sprite number switch case /if else statement. As the values not tested were impossible to generate through the natural game logic, we ignored these cases.

For coverage in the PlayerTest we were able to get line coverage to 34% and branch to 17%. This was due to the branches involving direction and many switch case or if/else statements. These values were not possible to test through game logic, so they were ignored. Additionally, some methods were private, so they were not added to testing.

For coverage in the object package we were able to get 75% line coverage and 38% branch coverage. This was due to the many if/else statements in the superobject draw method. As the values not tested were impossible to generate through the natural game logic, we ignored these cases.

For coverage in the utility, gamsetter, and playsound tests, we were able to get 100% line and 100% branch coverage.

For the sound test, we were able to get 88% line coverage and 100% branch coverage.

For the CollisionChecker and CollisionInstance test files, we were also able to bring line coverage up to 100%. We have 75% branch coverage again due to switch statements involving the direction parameter and we chose to not get full coverage as we would have to test with values that were impossible to generate naturally.

For the asset setter class, we got 98% line coverage and 90% branch coverage.

For the game console class, our coverage was 45% for line coverage and 17% for branch coverage. We didn’t test the paint component method since that is an internal run by jpanel. As a result, we were not able to get a higher line or branch coverage. However, much of the calls in the paintcomponent were tested in other packages.

The class that we did not test was screen settings as that class had static methods, which are not suitable for testing.

5. Findings

When writing these tests, we did make several changes to the existing code base. We discovered there were several areas that included dead code that was not being used. The use of code coverage made these issues more apparent as we began asking why our test cases were not covering certain parts of code and therefore made us look more critically at areas we previously thought were fine.

TileManager consists of 3 tests. One test checks a valid map and asserts that the output Tile[][] consists of valid instances of tiles. One test checks an invalid map consisting of numbers not corresponding to a type found in the Tile.Type enum, and asserts the generated tiles are null. The last test expects the resulting String[][] to be null and asserts such. The tests in TileManagerTest cover all code except one branch in setTiles() and the catch block in the readMap() method.

We found there were some initial issues in getting the maven jacoco code coverage plugin to work as intended. We discovered online that this is a known bug however, and can get around this issue by using the ‘mvn clean jacoco:prepare-agent install jacoco:report’ command to successfully generate a code coverage report.