Phase 3 Report

Group 20
Nov 25, 2022

Unit and Integration Testing

Some features we have tested are our BFS search algorithm, keyboard inputs, map and entity generations, and our tick system.

BFS is currently covered by eight tests, which test how the enemy chases the target in a variety of positions on the map. Four of these tests are used to test if the enemy correctly chases the target from one block away, and the four others test this from multiple blocks away.

There are two tests for keyboard input; one tests for a single arrow key press on the keyboard, and the other tests for multiple arrow key presses at once.

Currently there are 10 tests for the map. We make an initial map for testing and then perform these tests on it. One tests various starting positions of the player on the map, focusing on the boundary cases. 11 different positions are tested. Another test checks that the correct entity names are retrieved by the getEntityNameAt() function after the creation of an entity.

The next test checks the position of the player when the player tries to move into a wall. The test checks the player coming at the wall from all four directions, and we expect the player's position to not change if they attempt to move into a wall. With the next test, we check the behaviour of the player when they attempt to move into an empty space. Then, with the following test, we make sure that the game ends with the appropriate conditions if the player moves into an enemy.

The following three tests ensure the intended behaviour for when the diver collects various entities (coins, seaweed, and treasure chests). There is another test used to check the end condition when an enemy runs into a player. Finally, there is a test that verifies the player wins when they reach the exit.

Lastly, we have two tests for the Tick class. One test verifies correct behaviour when tick is incremented, and the other verifies it when tick is incremented and then decremented.

Test Quality and Coverage

<u>MenuTest:</u>

For our menu tests, we resorted to manual testing for navigating the menus and checking the display for different screens. We felt that the quantity of our code was small enough to test by hand and that it would be more efficient to physically go through the game ourselves. These manual tests also cover our ClickButton class as the game menus contain these buttons and they are required for the menus to properly function.

Line/branch coverage is difficult to quantify with manual testing as it's hard to know the exact cases being tested from an outsider's perspective. However, we are able to access each menu without issue through a variety of manual tests so it seems we have achieved mostly complete coverage.

All menus have been tested and are working without issues.

BFSTest:

To improve our coverage, we could add obstacles and other entities in the tests. Our BFS tests also test our position class because position is frequently checked in the BFS algorithm.

The main chunk of code being tested is the Search() function, which finds a path to the diver using BFS. This function consists of a while loop with an if statement and else statement, and several if statements nested in the if statement and else statement. Each inner if statement corresponds to the direction of the diver compared to the shark, and as our BFS test covers all directions we achieve 100% line coverage in this function.

Branch coverage is also 100% for the Search() function, as the code is composed of branch statements, of which we showed coverage in the paragraph above.

The results show that the shark is capable of using BFS to find its path to the diver in any direction, even multiple blocks away.

The map we generate for our BFS tests only includes border walls, the shark and the diver. That means that other entities like seaweed, treasures, coins and exits were not part of the BFS coverage.

KeyInputTest:

We were able to achieve one full line and branch coverage within the evenManager.java file. We covered the different possible branches by writing tests for when a single arrow key is being inputted, when no input is given and when multiple arrow keys are being inputted.

The results of the tests showed that the correct direction is being associated with the directional key input (eg. right arrow key = right). It also showed that when the user isn't pressing anything or if they are pressing multiple keys, the program behaves appropriately and doesn't move the diver.

We didn't test what would happen if an input other than the arrow keys was inputted, because there's no code that responds to that kind of input, meaning the diver would just stay still, which is the desired outcome.

MapTest:

Our Map tests also happen to cover much of our GameFactory and DefaultGameFactory code as they frequently interact with our Map code. We use the Map class to create game boards in these two classes, meaning what gets tested in our map tests also applies to these classes.

We were able to get full line and branch coverage for the Map.java file. We achieved this by creating different tests for specific functionalities of the map. Within the tests, we attempt to always test the different branches of possible outcomes as well as the components that lead to the outcomes. For example, when testing when the diver collects a

coin, the diver is put through a scenario where the coin that's collected is the final coin and when it isn't. The test also checks if the coins remaining had decreased, if the player score had increased and if the coin entity had been deleted, all components involved in the coin collection.

The results tell us that the map functions are behaving correctly, entities are moving as they should be, resources are being collected properly, scores are being updated and collection detection is accurate.

TickTest:

Unlike the other tests, the TickTest actually doesn't cover that much of the Tick.java file. It mainly just checks if the map is being updated every second and if the entity is being moved at the correct time as well.

The results show us that the tick function is working properly and updating at the right intervals.

The reason why most of the Tick.java file isn't being tested, is because a lot of the code borrow variables and functions from Map.java and we already tested that they work and update as they should. Meaning there's little reason to check if the functions work again in Tick.java but instead should just check if the map is being updated every tick.

Collectible, Coin, Diver, Entity, Shark, Wall, Exit Tests:

Like the menu classes, these classes had some manual coverage. We checked that various aspects of gameplay worked correctly, including that collectibles would give the correct game score, sharks would end the game if touched by the diver, and that the player could not move through walls. As with the menu tests, it is hard to quantify this testing in terms of line/branch coverage. However, everything tested manually is working as intended.

Findings

The main lesson we learned is that achieving complete code coverage is a big challenge. The examples we looked at in lectures were very small pieces of code, and although our program isn't very large it requires many more tests to achieve the same coverage.

The quality of code has improved and we were able to fix some bugs, and our current tests all pass. For example, when testing the BFS class, we found out BFS returns a direction array that doesn't contain the last direction. We fixed this bug by putting the last direction to the array before the return.

Our code is not perfect - in particular, there is still a lot of slowdown and high memory usage when the player gets too far away from the shark.