CMPT 276 PROJECT, PHASE 3
"UNDERWATER ADVENTURES"
GROUP 19

Tommy (Seahoun) Kim - 301400674
Hazelle Lebumfacil - 301391749
Jagdeep Singh -301540139
Tsu-Erh Yu - 301426092

Section 1- Unit and Integration Tests:

Unit Tests (Single Features in Isolation)

Entity

Feature 1: For any class that extends Entity, the x and y position of the object should be able to be set after creating an instance of the class. Testing covered in the file EntityTest.java

Maze

Feature 1: An object of type Maze can be set with a constructor with a mapGrid parameter that is statically defined in the class. This feature is tested in the method testSetMaze().

Feature 2: The mapGrid parameter can be indexed and edited. This feature requires a check that the mapGrid actually has changed once it has been set. This feature is tested in the method testChangeMaze().

Both features covered in MazeTest.java

Scuba Controller

Feature 1: Adds a Scubadiver object into a LinkedList, and then places the Scubadiver object onto GameBoard in its specified x-position and y-position by using Scubadiver Feature 1. Testing covered in file ScubaTest.java, in test case testSetScubaController().

Scubadiver

Feature 1: Given the specific positions, Scubadiver can be set onto GameBoard in its correct x-position and y-position. Testing covered in file ScubaTest.java, in test case setScuba().

Feature 2: Set a boundary of 40x40 around Scubadiver for interaction purposes. Testing covered in file ScubaTest.java, in test case testgetBounds().

Feature 3: Scubadiver's position can be set back to its default position after game restarts. Testing covered in file ScubaTest.java, in test case testDefaultPosition().

All features covered in ScubaTest.java

Shark

Feature 1: Given the specific positions, Shark can be set onto GameBoard in its correct x-position and y-position. Testing covered in file SharkTest.java, in test case setShark().

Feature 2: Set a boundary of 40x40 around Shark for interaction purposes. Testing covered in file SharkTest.java, in test case testgetBounds().

Feature 3: Change the speed of Shark according to its current x-position so it goes back and forth in its provided pathway. Testing covered in file SharkTest.java, in test case testSharkSpeed().

All features covered in SharkTest.java

Shark Controller

Feature 1: Adds Shark objects into a LinkedList, and then places the Shark object onto GameBoard in its specified x-position and y-position by using Shark Feature 1. Testing covered in file SharkTest.java, in test case testSetSharkController().

Squid

Feature 1: Set a squid in any position on the board. Testing should cover that the object has been created and it has an x and y position. Testing covered in file SquidUnitTest.java in test case testSetSquid.java.

Feature 2: In the Squid class, there is a method to set the 'touched' parameter for a squid to true. Testing should check that this squid's 'touched' parameter can be changed. Testing covered in file SquidUnitTest.java in test case testSetTouched().

Testing is covered in the file SquidUnitTest.java

Squid Controller

Feature 1: When an instance of the SquidController object is created, it automatically creates two preset Squid on the board. The testing should ensure that both are present. Testing covered in file SquidUnitTest.java in test case testSetSquidController().

Feature 2: The squid controller has an arraylist to keep track of the squid present on the board. Check to see if new objects can be added to the list. Testing covered in file SquidUnitTest.java in test case testAddSquidToList().

Feature 3: If a squid is touched, the squid controller will go through the list to ensure that the correct one was set to 'touched'. Testing covered in file SquidUnitTest.java in test case testSquidFromListTouched().

Turtle

Feature 1: Set the turtle on the board with its specified x-position and y-position. Testing should cover that the turtle has been created in its specified x-position and y-position. Testing covered in file Turtle test in test case testConstructor().

Feature 2: Make the turtle move by pressing the WASD key which changes the boolean values upPressed, downPressed, leftPressed, and rightPressed to true. Testing should cover that the turtle's position changes when moved to the according direction. Testing covered in file TurtleTest.java in test case testTurtleMove().

Bonus Rewards

Feature 1: Set worms in the random position and check whether the worm is on the MapGrid. Testing covered in file BonusRewardTest.java in test case setSWorms().

Feature 2: Set shrimp in the random position and check whether the shrimp is on the MapGrid. Testing covered in file BonusRewardTest.java in test case setShrimps().

Regular Rewards

Feature 1: Set keys in the random position and check whether the key is on the MapGrid. Testing covered in file BonusRewardTest.java in test case setShrimps().

Integration Tests (Features Tested Together)

While running maven on automated tests, there was an issue we discovered with creating multiple instances of the GameBoard class across multiple testing files. Therefore, integration tests were done with features that depended on the Maze class to mitigate this issue. Additionally, there were few test cases that succeeded when run individually, however failed when run with all the other tests together, which resulted in us removing them.

Turtle + Bonus

If the turtle and bonus rewards are in the same cell, the score should go up. If it is in the same cell as a worm, the score should go up by 10. In the same cell as a shrimp, the score should go up by 20.

The score will only get updated in the game board

Turtle + rewards

If the turtle and keys are in the same cell, the score should go up by 20, and the number of keys collected should go up by 1.

The score will only get updated in the game board

Turtle + shark

If the turtle and the shark are in the same cell, the state of the game should automatically switch to the game-over state. Testing covered in file SharkTest.java, test case testSharkInteract().

Turtle + scuba

The scuba diver should be able to move as close to the turtle as possible based on their relative distance. If the turtle is above the scuba diver, the scuba diver will move up. Similarly, if the turtle is either on the right, left, or underneath the scuba diver, the scuba diver will move accordingly. Testing covered in file ScubaTest.java, test cases testScubaTrackTurtleUP(), testScubaTrackTurtleDOWN(), testScubaTrackTurtleLEFT(), testScubaTrackTurtleRIGHT().

Turtle + barrier

The turtle should not be able to move through barriers set on the maze. This feature should check that the turtle is not making any illegal moves by going through barriers. Testing covered in MazeInteractionTest.java, test case testTurtleInteract().

Turtle + squid

If the turtle and the squid are in the same cell, that should trigger the squid to switch its ‘touched’ parameter from false to true. There are multiple squid on the board so these tests ensure that only the touched squid is set. Testing covered in file SquidUnitTest.java, test case testSetTouched().

Scuba + shark

The scuba diver is programmed to get as close to the turtle as possible based on their relative distance, but the scuba diver should not be able to move through the pathway of Shark set on the maze. This feature should check that the scuba diver is not making illegal moves to get closer to the turtle. Testing covered in file ScubaTest.java, test case testScubaSharkBoundInteract().

Maze + Entity

If the maze is edited (for example, changing a barrier to empty or vice versa), the entity should be able to react to the change. If a new barrier is set, the entity should not be able to move through it. If a barrier is taken away, the entity should be able to go through. Testing covered in file MazeInteractionTest.java, test case testBarrierChangeTurtle().

Maze + Turtle

The Turtle should not be able to move through barriers set on the maze. Since the turtle's movements are dependent on the MapGrid from the Maze, it should also check that a keypress doesn't allow the turtle to go through a barrier. Testing covered in MazeInteractionTest.java, test case testTurtleInteract().

Maze + scuba

The scuba diver is programmed to get as close to the turtle as possible based on their relative distance, but the scuba diver should not be able to move through barriers set on the maze (similar to the turtle). This feature should check that the scuba diver is not making illegal moves to get closer to the turtle. Testing covered in MazeInteractionTest.java, test case testScubaInteract().

General Performance Testing

To ensure that there were no bugs that were unforeseen by the unit and integration tests, the team played the game many times to find issues in the production code. These issues are further detailed in the conclusion section of the report. This testing acted as our regression testing, ensuring that after we fixed the faulty features, those fixes did not affect the existing production code.

Section 2 - Test Quality and Coverage

To ensure quality in our test cases, our group took the following measures:

- Communicated frequently for integration tests to guarantee that there were no redundant test cases / repeated cases
- Made sure that tests were successful and ran both through the Visual Studio Code User Interface and through running maven commands through the terminal
- Had a document that acted as a test plan to ensure that at least the basic features of each class were being tested

Line and Branch Coverage:

Typically, with maven automation, you would be able to use the jacoco plugin to get coverage reports, but upon experiencing issues with our pom file, the plugin was omitted from our final tests.

Formally, the line coverage is computed as the [(lines covered) / (total lines)] * 100%, but we recognize that the line coverage is not the most effective way to test adequacy, since the number of lines will be case by case for every programmer.

Our test cases were similar to a blackbox testing model, where we were mostly focused on the functionality of each class and its features.Therefore, our test cases had approximately 60% coverage overall within all of our files.

Formally, the condition (branch) coverage is calculated as (outcomes covered) / (total conditions) * 100 %. Our conditions are covered in the integration tests, while also ensuring that there is no redundancy. An example of a conditional test would be checking if the scuba diver can move to an empty cell or a cell that contains seaweed depending on if there are barriers blocking it. We aim to get close to 100% for condition coverage and given that our tests could not cover gameboard tests, we had approximately 70% condition coverage overall, and closer to 80% condition coverage for individual features and unit tests.
Some of the code in the Gameboard class are unable to be covered in testing, as we experienced an issue upon creating multiple instances of objects from the Gameboard class causes issues with the maven automation.

Section 3 - Unit and Integration Tests:

What have we learned from writing and running tests:

- For large scale tests, it's hard to determine what could be an issue because it can't pinpoint where an error might exist
- Organizing your tests are really important, especially for integration tests since you don't want to be redundant
- Some tests were repetitive with different parameters

Revealed Bugs:

- Fixed the lag from the production code, the game runs much smoother than it did before.
- Fixed the issue where the keys weren't resetting when a player preemptively died
- Fixed the issue where the squids weren't appearing again after interaction with Turtle and restart happens

Changes to Production Code:

- Included cleanup and restart functions for keys and squids upon restarting the game
- Fetching images from resources at every game loop made the game slow. Therefore, we stored the fetched image in a variable and used it for redrawing.
- Cleaned functions to be more generalized, tried not to hard-code where necessary
- Included getter for speed of Shark