CMPT 276: Phase 3 Report

Concept Refresher

The game we chose to implement was a 2D pixel-style game called “Monkey Escape”. In this game, the user controls the monkey protagonist whose goal is to collect keys to escape the zoo. During the escape, there are numerous opportunities to gain points by collecting bananas, but the player must be cautious to not get caught by the zookeepers.

Overview of Testing

To organize the testing phase of our project we followed a similar strategy to the agile scrum methodology we utilized during the execution of Phase Two. Using our team availability schedule we collected during Phase Two we were able to plan our sync-meetings for every second day. To limit the overlap of testing responsibilities, we assigned each group member a section of code that they were responsible for testing. We ensured that each member was only responsible for testing code that they did not make major contributions to during the development stage. This decision was made as it is often easier to overlook logic gaps and other errors in code you wrote or are familiar with. We organized our testing so that each of our implemented classes has a testing class with the naming convention <Class name>Test.java. We tested abstract classes and interfaces through the various concrete class dependencies. Organizing the testing this way allowed us to clearly define unit tests for each class, and track interaction based tests in the corresponding testing suite.

Tested Features

As mentioned above, we chose to organize our unit tests on a class by class basis. Similarly, any integration/interaction testing between classes was written under the testing class we are checking the interaction and calling assertions for. A brief explanation of all unit testing and integration testing is outlined below:

CLASS: Banana.java
UNIT TESTING: Since this class was largely dependent on other modules to support functionality, the majority of the testing was done on FixedEntity functions. However, the access modifier functions, *getLifecycle() and setLifecycle()*, were tested with various inputs to ensure that we could get and set the lifecycle class variable if needed externally. The *update(), remove(), and playSound()* methods primarily just call functions from other classes, so there was minimal logic to test. However, in *update()* we did check the conditional statement executed as expected for both positive and negative lifecycle values.

INTEGRATION TESTING: The majority of the integration for this class was covered under unit testing for the FixedEntity and Banana class inheritance interaction. To test this relationship we performed testing of the FixedEntity functions through a Banana class object. We also tested any external class variables that were changed through the methods in the calling class, such as the panel class.

CLASS: FixedEntity.java
UNIT TESTING: This class is an abstract class, and thus had the majority of its testing completed through its concrete children classes. The *loadImage()* function is a largely graphics-based method, so we were only able to test that the method asserted when presented with an invalid image. The *createRandomPosition(Panel panel)* function tested that for various panel configurations that a valid position within the wall boundaries was created.
INTEGRATION TESTING: The majority of the integration for this class was covered under unit testing for the FixedEntity children class interaction. We also tested any external class variables that were changed through the methods in the calling class, such as the panel class.

CLASS: Key.java
UNIT TESTING: Since this class was largely dependent on other modules to support functionality, the majority of the testing was done on FixedEntity functions. These methods were tested as stated in the FixedEntity section.
INTEGRATION TESTING: The majority of the integration for this class was covered under unit testing for the FixedEntity and Key class inheritance interaction. To test this relationship we performed testing of the FixedEntity functions through a Key class object. We also tested any external class variables that were changed through the methods in the calling class, such as the panel class.

CLASS: LionPit.java
UNIT TESTING: Since this class was largely dependent on other modules to support functionality, the majority of the testing was done on FixedEntity functions. These methods were tested as stated in the FixedEntity section.
INTEGRATION TESTING: The majority of the integration for this class was covered under unit testing for the FixedEntity and LionPit class inheritance interaction. To test this relationship we performed testing of the FixedEntity functions through a LionPit class object. We also tested any external class variables that were changed through the methods in the calling class, such as the panel class.

CLASS: Monkey.java
UNIT TESTING: The Monkey class solely focuses on the movement of the monkey/player using *update()*. Therefore the monkey's movement was tested using all different directions and whether or not it is stuck in a lion pit.
INTEGRATION TESTING: The *update()* method in the Monkey class relies on the collision detection in the Collision class for if it can move, and the game's State for if it should move. The collision detection was tested in the Collision class testing, but the *update()* method was tested with the game being in Play mode and not in Play mode.

CLASS: Zookeeper.java
UNIT TESTING: For the zookeeper class, the features that have to be tested are *update* and *searchPath.* These are essential methods that contribute to the movement of the zookeeper. The *createRandomPosition* method didn't have to be tested because it was tested in MonkeyTest.java. The *drawImage* method wasn't tested because there is no clear way to test whether or not the image was drawn.
INTEGRATION TESTING: The *update* method in the Zookeeper class was integrated in the *update* method of the Game class, and the *searchPath* method was integrated in the Zookeeper *update* method.

CLASS: Pathfinding.java
UNIT TESTING: The pathfinding class was responsible for implementing the pathfinding algorithm for the zookeeper to find the monkey. The main features that have to be tested are the *instantiateNodes*, *resetNodes*, *setNodes*, *getCost*, *search*, *openNode*, and *trackThePath* methods.
INTEGRATION TESTING: As mentioned above, all of the pathfinding methods were integrated in the methods of the Zookeeper class.

CLASS: Collision.java
UNIT TESTING: The Collision class is responsible for detecting collisions on the map using it's methods *checkTile()*, *checkFixedEntity()*, and *checkZookeeper()*. These methods call on the function *processCollision()* to process the possible collision. *checkTile()*, *checkFixedEntity()*, and *checkZookeeper()* were each tested individually for inputs including invalid positions and positions where there was collision or was not a collision. *processCollision()* was tested on an empty list of collisions, collisions with a lion pit, and collisions with other fixed entities.
INTEGRATION TESTING: The integration tests for the Collision class were done in the unit testing of the moving entity classes which use collision detection for their movement.

CLASS: Game.java
UNIT TESTING: The Game class is responsible for the spawning of entities and setting up the game at restart, and next level. The spawn methods were tested to ensure that the entities were created and added to the list of entities. The *restart()* method was tested by asserting that the list of entities, list of zookeepers were cleared, the level was reset to 1, score was reset to 0 and timer was reset to 0 when called. The *nextLevel()* method was tested by asserting that the level was incremented by 1 and that the list of entities were cleared and the initial entities were respawned and added to the list of entities and had the number of zookeepers were the same as the level. The *update()* method was tested to ensure that the state was changed based on the score and that the restart state was detected.
INTEGRATION TESTING: The *makeNewMap()* method called by *restart()* and *nextLevel()* was tested by the unit tests of the TileMap class. Interactions between entities were tested by the unit tests of the various entities.

CLASS: KeyHandler.java
UNIT TESTING: This class is responsible for processing KeyEvents raised by the Panel and used by classes that required key input. The *keypressed()* method was tested by passing a KeyEvent for a key press and checking that the appropriate boolean was being set to true and that any other booleans were not being set to true. *keyreleased()* was tested similarly but for setting the appropriate boolean to false.
INTEGRATION TESTING: The integration tests for this class were done by Unit tests of the classes that required key input such as the Monkey and State classes.

CLASS: Panel.java
UNIT TESTING: This class is responsible for drawing to the screen, running the game loop and render loop as well as keeping track of the time. The unit testing done for this class was ensuring that the correct number of seconds was added to the timer.
INTEGRATION TESTING: The integration test for this class was done by the Window class for adding a Panel to the Window.

CLASS: Sound.java
UNIT TESTING: The sound class is responsible for playing the background and interactive sounds of the game. It wasn't really possible to find out if the sound is actually playing, so we couldn't find a way to test the *play* and *loop* methods. It was also not straightforward to test the *setFile* method, so we just tested to validate that the sound file is never null.
INTEGRATION TESTING: The *setFile* and *play* methods were integrated in the entity classes that have a sound being played (Key, Banana, Game). The *loop* method was integrated in the Game class where we loop the background music.

CLASS: State.java
UNIT TESTING: The state class is responsible for handling the state of the game (pause, play…) and determining what the next state should be. The method that has to be tested is the *changeState* method, and the test cases should cover all of the different states.
INTEGRATION TESTING: The changeState method was integrated in the Panel class when a key was pressed to process what state the game should change to.

CLASS: Window.java
UNIT TESTING: The Window class only has one method which relies on the integration of the Panel class so there was no unit testing done for the Window class.
INTEGRATION TESTING: The Window class only has one method, which adds a panel to the window. This method was tested by adding a new panel to the window and checking the component count on the window before and after.

CLASS: MapGenerator.java
UNIT TESTING: The MapGenerator class has three methods that work together to create a new map that is used for that level in the game. These methods are the *generateRandomMap(), newRandomMap()*, and *addSpaces()* methods, and they were all individually tested for their purposes.

INTEGRATION TESTING: The integration tests for this class were done by Unit tests of the TileMap class which uses the MapGenerator class in the creation of new levels.

CLASS: TileMap.java
UNIT TESTING: The TileMap class is used to make the map and add fixed entities to the map using the *getTiles()*, *makeNewMap()*, *generateMap()*, and *addFixedEntityToMap()* methods. *getTiles()*, *makeNewMap()*, and *generateMap()* were tested individually for their purpose in the process of making a map. *addFixedEntityToMap()* was tested by inputting a FixedEntity then checking the map if it was added.
INTEGRATION TESTING: The integration testing for TileMap was done in the Panel class which uses it as a map for the game.

CLASS: Painter.java
UNIT TESTING: The Painter class was composed of mostly graphical methods which we were not able to find an effective way to test. The method that was tested was the *getImage()* method which was responsible for getting the image for the menu. The getImage() class was checked to ensure that an image was being returned for each menu type.
INTEGRATION TESTING: Since the methods were mostly graphical, we were not able to test integrations with other classes.

Test Quality and Coverage

Similarly to phase two, one of the main measures we enforced to increase the quality of our code was mandatory code reviews/merge requests. We also ensured that the class tester was different from the class developer for each class to ensure that nothing was overlooked and there were no biases from the class developer. During the testing process we aimed to have regular discussions between the class developer and class tester. We did this as it encourages a more thoughtful review process of our current code base and facilitated a more seamless debugging process when necessary. In general, we aimed for 100% branch coverage, ensuring that all of our conditional statements were executed. However, since we had a lot of class interactions and calls to external libraries we were not able to achieve 100% line coverage, but we got as close as we could reasonably get. While we aimed to test all functionality in our code, we could not determine an effective method for testing graphical methods. Since the majority of this would be comparing produced visuals against our desired visuals, we mainly tested any assertions and concrete behavior present in these methods. Unfortunately this left a large portion of our graphic-based functions untested for desired functionality.

Test Findings

One of the main things we learned from writing and running our tests is the importance of developing code with minimal coupling. Since writing our unit tests required us to test each functionality in isolation, we discovered that there were occasionally dependencies that had developed during the implementation that made it difficult to isolate one class from another. A prime example of this is the relationship between our Panel and Game classes. The behavior and functionality of these two classes were not easily separated, and thus made it difficult to test. This situation created a discussion behind the purposes and responsibilities of each class, and resulted in a slight refactorization of these two classes. We settled on denoting the responsibility of the UI and interaction to the Panel class and the Game model to the Game class. We moved Game logic such as nextLevel() and restartLevel() that were in the Panel class to the Game class.

We found a bug in the Sound class while testing. The sound class takes in an index for a sound file, and sets the sound clip to that audio file with the given index. We found that when an invalid index was given, the clip would be null, which then would cause a null pointer reference when it is referenced in the play and loop methods of the sound class. We solved this by checking if the index is out of bounds before setting the sound clip. A similar bug was found in the TileMap class and the Collision class when accessing a position on the map, there was no check beforehand that the position on the map was a valid position before checking for it in the array. This would cause the code to crash when checking for an invalid index in an array.