CMPT 276 Phase 3 Report

2.1 Unit and Integration Tests

Identify the features that need to be unit tested, and briefly explain these features in your report. Next, identify important interactions between different components of your system (e.g., the logic and the user interface/file system/etc.). Explain such interactions in your report, and create integration tests for them. Document which test case/class covers which feature/interaction in your report.

We decided to partition the tests depending on specific aspects of the game we wanted to test. These partitions are in different files called CharacterTest.java, EnemyTest.java, KeyControllerTest.java, SoundTest.java and UITest.java. For every test file we have a setUp and reset function that initializes the needed objects and variables for the tests, then resets them after the tests complete. The specific tested features and test cases are explained below.

CharacterTest

CharacterTest.java holds the CharacterTest class which contains all of the tests pertaining to the main character's functionality and its interactions with other things in the game. The most important features of the character we wanted to test were its movement, collision detection, interactions with enemies, interactions with the rewards, interactions with the walls/environment, and that its image was being drawn and updated properly. In order to do this, we initialized the necessary character, enemy and game objects in the setUp function. Afterwards we applied specific keyboard inputs or created specific scenarios for which we could assertTrue for a variable that the Character class held. For example, to test movement we would simulate a keyevent such as a "w" press and check if the player's Y position increased by the intended amount. We would then do this for all the directions. Another example is for testing collisions. We would place a reward, or enemy right next to the character and see if the character's score would go up or down by the intended amount. We would then do this for different rewards, traps, enemies and even the walls. After each test the objects we initialized would then be set to null or reset.

EnemyTest

EnemyTest.java holds the EnemyTest class which contains all of the tests pertaining to the ghost and zombie functionality. The most important features we wanted to test were the traversal/tracking of enemies around the map and to the main character, as well as their collisions with one another. Similar to the character tests, we initialize the game object and enemy objects in the setUp function, and set them to null in the reset function. In order to perform the traversal/tracking tests, we placed a main character and an enemy at specific locations on the map. Afterwards, we would loop the enemy update function in order to simulate its movement over time. Then, we check if the player has taken damage, if it has then the enemy has successfully tracked otherwise it has not. We would then do this for the different types of enemies since the zombies use a ray tracing algorithm, while the ghosts use a calculation based on current position and character position.

SoundTest

Soundtest.java holds the SoundTest class which contains the tests pertaining to the sound generation in our game. The main features we wanted to test were the music that plays during the game, and the sound effects generated when you run into enemies or collect rewards. In order to do this, we would once again initialize the needed objects such as the character, enemy and rewards. Afterwards, we then create scenarios where a specific sound object would be initialized and assert true for the existence of this object. If it existed, the functionality was correct, otherwise it was not. For example, in order to test the reward collection sound, we would place the character on top of a reward, then assert true for the generation of the sound object. We also did this for collisions with enemies and the case where the player was just stationary in game. We also tested if our musicStop() function worked by checking if the sound objects were properly set to null.

UITest

UITest.java holds the UITest class which contains all of the tests pertaining to the UI, title screen, game screen and the drawing of objects. This is one of the most important aspects of the game as it is the interface the users use to interact with the game. However, it was also the hardest to create tests for as "visual features" are most easily tested by actually running the game. Nevertheless, the most important features we wanted to test was the generation of the different screens such as the pause screen and title screen. We also wanted to ensure the generation of the map images, map walls, objects, enemies along with their movement animations. In order to do this, we once again created the game object and asserted true for specific variables that we knew should be initialized. For example the pause screen, after the player presses "p", a pause screen image object should be drawn to the screen. In order to test this, we simply just check the existence of this image object. If it exists the functionality is correct otherwise it is not. We did this same test for a variety of other images such as the title screen, game screen (map), and for the characters' animations.

KeyControllerTest

KeyControllerTest.java holds the unit tests for KeyController class which tracks key pressed, and keyreleased using keylistener extended class KeyController. The test cases test the SimulateKeyPressed() method in KeyController class. Keystroke W, S, A, D, P, and Enter was tested where W, S, A, D changes boolean variables upPushed, downPushed, leftPushed, and rightPushed. Keystroke P was tested to validate game state changes from play state to pause state. Lastly, Keystroke Enter in title state changes the game state to play state if only if the cursor is on the new game selection.

2.1.2 Test Quality and Coverage

Discuss the measures you took for ensuring the quality of your test cases in your report. Code coverage has been traditionally used as a popular test adequacy criterion. Measure, report, and discuss line and branch coverage of your tests. Document and explain the results in your report. Discuss whether there are any features or code segments that are not covered and why.

As mentioned above, our test cases were designed to partition parts of the game and test core implementations of each partition used in our game.

For example, our CharacterTest.java tested the main characters movement, collision detection, interactions with enemies and rewards. To improve line and branch coverage we tested idle state, move state (up, down, left, and right), collision event, non collision event, invalid collision with rewards, valid collisions with reward, enemy collision, and wall collision. These test cases for the CharacterTest.java covered most of the code line-per-line except for catch statements and draw() function. More specifically, catch IOexceptions during IOImage read, and draw Graphics2D images drawing different character PNG files to simulate animated movement based on System.nanotime() function. Because testing for PNG image file not loading is redundant as catch IOexception catches such occurrences. Additionally, checking for which png file was loaded based on nanoseconds seems inadequate as well as not loading the correct PNG file would also be caught in the catch block for IOexceptions. Overall line and branch coverage for Character.java was 93.6% and 84.3% respectively.

For enemy class tests, we tested primarily the character tracking methods for our two enemy types (zombie, and ghosts). Collision test was redundant as collision test with players was done in the character test. Similar to character test, line, and branch coverage for our enemy test covered most of the codes except for, again, IOexception and enemy direction which changes PNG accordingly. Overall line and branch coverage for Zombie.java, and Ghosts.java are 82.6-92.3% and 74.1-76.7% respectively.

For the sound test, we tested if the music plays when the update() method for the game is performed. Additionally we tested if the much turns off correctly when the stopMusic() method is called. As before each test was performed with a setup() and reset() to test each test case independently. Line coverage for Sound.java is 100% and branch coverage does not apply as there are no switch statements or loops in Sound class.

To compensate for IOexception in the character class test, and to follow the partitioning practice used in our unit tests, we tested the UI class for our game in different ways. We tested if the game instance starts correctly, the draw() method with Graphics2D instantiated draws generic images correctly, such as pause state and title screen state, and lastly, we tested if the UI class correctly draws all the character sprite PNGs. Because UITest.java tests for drawing of all the sprite png images for the main character we indirectly tested for the lines missed in character unit testing. Overall the line, and branch coverage for UI.java 88.6% and 50% respectively. Branch testing for UI class was low because it did not test for messages drawn during play state where the player is shown a message when collecting rewards. The main reason for not testing for such a case is because our camera for the game follows the player, hence relative, and the messages also being relative in absolute positioning.

Lastly, the KeyControllerTest.java tests the KeyController class in our game. Keystroke W, S, A, D, P, and Enter was used to test SimulatekeyPressed() method. As mentioned previously, test cases were set up to make sure boolean variables activated when keystrokes are detected using KeyController class. Furthermore, P and Enter keystrokes were tested to check if game states change to play, pause from play state and to play state from title state. The line, and branch coverage for the KeyController class were 39.9% and 33.3% respectively. Our test cases show low line, and branch coverage as testing because code segments for KeyController were intentionally skipped. Our KeyController class is made up mostly of if statements for its code. Two methods keyPressed(), and keyReleased() were intentionally left out of testing because testing for these methods which uses parameter KeyEvent, a java library import from Keylistener required out-of-scope knowledge that requires further studying on an implementation redundant for testing purposes. As our test cases already test for keystroke input W, S, A, D, P , and Enter, and test the correct switching of game states. Thus further testing for keyPressed and keyReleased methods show redundancy as SimulatedkeyPressed validates keystroke inputs.

Unit test results for our entire game show 86.8% for line coverage, and 61% for branch coverage. Majority of loss of coverage comes from switch statements used to draw PNG sprite images for characters and enemies which are subclasses of Animate class (where the loss of coverage resides), and from KeyController class mentioned above where test cases were intentionally skipped as it was deemed redundant.

2.1.3 Findings

Briefly discuss what you have learned from writing and running your tests. Did you make any changes to the production code during the testing phase? Were you able to reveal and fix any bugs and/or improve the quality of your code in general? Discuss your more important findings in the report.

Writing different test cases for the game components such as Character, Enemy, Sound, UI and key controller provided a proper understanding of their individual functionalities and interactions within the game. Each test case provides a deeper knowledge into how each of these elements behave under specific conditions, aiding in a deeper comprehension of their roles and behaviors within the game's environment.

Analyzing line and branch coverage across the test cases provided crucial insights into the thoroughness of the testing scenarios. It offered a clear understanding of areas where the tests were robust and those that required more attention. Recognizing and acknowledging low coverage areas facilitated a more focused approach to improving the testing suite's comprehensiveness.

Testing process revealed specific areas with low coverage, notably in image rendering for characters/enemies and segments of the KeyController class. These insights emphasized potential gaps that could affect the testing suite's overall effectiveness. Understanding these areas of limitations is important for the future efforts to increase coverage in these crucial aspects of the game.

In depth testing of character movement , enemy behavior, and UI generation showed us potential failure points and helped in identifying areas where improvements or optimizations could be made to enhance overall performance and reliability.

In summary, the testing phase was extremely important in providing comprehensive insights into the game's elements, revealing potential bugs, and improving the overall quality of the code. Addressing uncovered issues, low coverage areas will likely contribute to more robust and reliable games.