CMPT 276: Introduction to Software Engineering

REPORT PHASE 3 - GROUP 09

Minh Phat Tran	301297286
Chun Hei Yiu	301329448
Steven Xia	301444281
Miles Burkholder	301405683

I. Introduction

In this phase, our group created a testing process for our game project “Easter Bunny Hunt”. We applied unit tests, integration tests and manual tests to test singular features of the game as well as the whole game by using JUnit, AssertJ Swing and JaCoCo. The reason to use JUnit, an automated test framework, is to create unit tests and integration tests automatically. Furthermore, AssertJ Swing is an open-source library that will make an automated Robot class that stimulates user input to play and control the game character. Swing GUI and game window are tested by AssertJ Swing as they are Swing classes (JPanel and JFrame), which will let developers track the loading process of input images whether correct or not. Our group will use Java code coverage tools (JaCoCo), which checks the testing coverage of our test case, the information of coverage percentages such as the instruction coverage and branch coverage of the program will be given via an HTML file called “index.html”, which will be generated in the directory “target/site/jacoco” after maven test. It can be accessed via browser as shown in figure 1. Currently, our test cases have an instruction coverage of 90% and branch coverage of 85%.

EasterBunnyHunt > default

default

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		77%		n/a	2	4	6	21	2	4	0	1
Maze		86%		79%	31	109	61	344	7	31	0	1
GameObject		89%		n/a	0	2	2	12	0	2	0	1
Maze.Key		94%		80%	7	26	4	60	0	4	0	1
Hero		96%		100%	1	15	1	24	1	8	0	1
Environment		100%		n/a	0	1	0	2	0	1	0	1
Punishment		100%		n/a	0	1	0	2	0	1	0	1
Cell		100%		n/a	0	2	0	3	0	2	0	1
SpeedBonus		100%		100%	0	4	0	6	0	3	0	1
ThornBushPunishment		100%		100%	0	4	0	6	0	3	0	1
FreezeBonus		100%		100%	0	4	0	6	0	3	0	1
ScoreBonus		100%		100%	0	4	0	6	0	3	0	1
TrapPunishment		100%		100%	0	4	0	6	0	3	0	1
Door		100%		100%	0	8	0	12	0	6	0	1
Wall		100%		100%	0	5	0	11	0	3	0	1
Bat		100%		100%	0	5	0	12	0	2	0	1
Enemy		100%		90%	1	9	0	13	0	2	0	1
Hunter		100%		100%	0	7	0	16	0	2	0	1
Egg		100%		87%	1	10	0	22	0	6	0	1
Character		100%		n/a	0	13	0	20	0	13	0	1
Wolf		100%		96%	1	21	0	31	0	3	0	1
Total	311 of 3,153	90%	41 of 290	85%	44	258	74	635	10	105	0	21

Figure 1: Percentage of instructions and branches coverage

II. Unit test

In this project, most features of our game will be tested in isolation via designed unit tests, and this can be done with the assistance of default functions JUnit assertEquals, assertNotEquals or assertNotNull.

In BatTest.java, our group has created a constructorTest function to test the initial position of the bat enemy when the game starts. Besides, the loading image represented for bat can be tested whether it loads correctly or not by function getImageTest. We also created the nextMoveTest_Turn function for this bat object, and it checks that the nextMove function changes directions every once in a while, which means deltaX and deltaY values are not set to zero before these tests. As a result of this, we got 100% for both instructions and branches coverage.

HunterTest.java has a similar constructorTest function and nextMoveTest_Turn with BatTest.java to test the location initialization and loading image process as mentioned in BatTest.java above. Now, we also test the random features of hunters via the function of nextMoveTest_Random. This function will check that the nextMove function of this object can make the character move in any of four cardinal directions, which means deltaX and deltaY values are set to zero before testing this to induce more variation because of randomness. According to the JaCoCo statistical data, we also got 100% for instructions and branches coverage.

Another kind of enemy class we created is WolfTest.java. This also includes constructorTest, getImageTest, nextMoveTest_Turn and nextMoveTest_Random functions, which are pretty similar to BatTest.java and HunterTest.java's ones as shown above. WolfTest.java has a function called nextMoveTest_Target which is used to check that the nextMove function of wolf sets the values of deltaX and deltaY to move it forward and follow the hero character, Bunny in our game. This kind of movement can be considered as an artificial intelligent movement of the wolf enemy. Therefore, we got 100% for instructions coverage and 96% for branches coverage.

As we have one hero in our game which is bunny, so we have created a HeroTest.java. In this HeroTest.java, we will create a basicTest function to generally test the getter, setScore and isDead of the bunny character. Besides, we also test the getImage function to make sure the loading image process is correct for the bunny. As a result of this, our instructions coverage is 96% and branches coverage is 100%.

The remaining testing files include CellTest.java, DoorTest.java, EggTest.java, FreezeBonusTest.java, ScoreBonusTest.java, SpeedBonusTest.java, ThornBushPunishmentTest.java, TrapPunishmentTest.java. All of them have a getImageTest function to check the loading images for each object in the game. We got 100% for all in terms of instructions coverage and branches coverage.

We recommend opening the "target/site/jacoco/index.html" after running "maven test" for a more detailed inspection of our test coverage.

III. Integration test

Our integration tests focus on the proper interactions between classes, this is all accomplished for us within the maze class which creates all the environment and character objects while making them interact with one another. By running a unit test of the maze it begins the game thereby encompassing integration tests between objects. Since the maze class creates many objects and relies on keyboard input we were unable to use many assertions within the unit test, instead, we resorted to functional specification-based testing. We created a Robot object from the Assertj Swing library in order to simulate keyboard inputs used for running the tests. Before each test was performed, a new game was run.

gameScreenTest:

This test tests the game states by pressing the space key, passing through the intro screen and the instructions screen.

pauseScreenTest:

This tests the pause screen once the game has begun, it resumes the game and pauses the game again. This allows us to confirm the different game states and their impact on all the character objects currently on the screen correctly responding and don't perform their next move while the game is paused.

unpauseTest:

This test un-pausing the game allows us to confirm that the character classes resume motion and it proceeds to pause the game again and exit the game early instead of resuming.

characterMoveTest:

In this test, the robot moves the rabbit in all 4 directions, attempts to collide with a wall environment object but is unable to continue the motion. This test also covers regular egg collection, the rabbit moves onto an egg object and as expected the egg disappears and the user's score is incremented by 1.

thornTest:

This tests the thorn bush point deduction interactions with the rabbit. It first collects an egg gaining a point and then runs into a thorn bush deducting a point resulting in a score of 0.

deathTest:

This tests the death interactions with the thornBush, the rabbit first moves into a thorn bush object while having a score of 0, once the rabbit collides with the thorn bush, it deducts a point from the score, the maze class then recognizes that the score is negative and the game ends. It also tests first collecting an egg and then running into 2 thorn bushes again resulting in a negative score thereby ending the game.

trapTest:

Finally the rabbit walks into a trap resulting in the trap correctly disappearing off the screen, a trap timer display appearing, and the robot being unable to move the rabbit until after the timer has gone to 0. Once the timer is up the rabbit moves freely again.

In conclusion, to perform these tests we created a special map isolating the rabbit from other enemies so the tests did not end prematurely. Although the robot allowed us to test many of our game's specifications through keyboard simulation, we were unable to test some functionality due to its variability such as collecting randomly generated bonuses', winning the game, and dying to individual enemies. The maze tests were able to achieve 86% instruction coverage, and 79% branch coverage, this is sensible considering it was not able to automate random object cases like Enemy Class and Bonus class that behave randomly.

IV. Manual test

Manuel black box specification-based tests, while time-consuming and inefficient, were conducted according to the specifications that could not be tested during integration testing to ensure that all requirements were met and to complete program verification. We could not test some specifications during integration testing because some features were not possible to automate due to random unpredictable behaviour. Behaviour such as enemy movements and their interactions, bonus generation and winning the game was too challenging to automate because they are different each time the game is played. The manual tests and their results are listed below.

Hunter test:

The hunter was not impacted by static environment objects such as eggs, traps, bushes, and bonus eggs. The bat's movements, while random, they were unable to move through a wall object (tree) or door object (portal) from any side. It was tested that the bat kills the user upon collision. Upon colliding with a wall the hunter changed directions.

Wolf test:

Testing movement as well as character interactions, the wolf is not impacted by other static environment objects and is free to move on traps, eggs, bushes, and bonus', the wolf was not able to move on wall (tree's) objects or on the door (portal). The wolf tracks and moves towards the user's position, this was tested by the user staying stationary and the wolf successfully moved positions toward the user and killed them. The wolf collided with walls from all directions and was unable to move through a wall.

Bat test:

Similarly, the bat was not impacted by static environment objects such as eggs, traps, bushes, and bonus eggs. The bat's movements, while random, they were unable to move through a wall object (tree) or door object (portal) from any side. It was tested that the bat kills the user upon collision.

Bonus insertion test:

It was observed that bonuses were appearing at random locations of the map and always on an empty cell containing no other environment objects. Once a bonus disappeared it would respawn again at a new empty cell after 10 seconds. We also Tested and confirmed that each time a bonus appears in the game it is randomly chosen to be 1 of the 3 available bonuses'.

Bonus deletion test:

It was confirmed that after the bonus has been on the game screen unclaimed for 10 seconds it will disappear. The bonus also disappears upon touching the user.

Freeze bonus test:

Upon collecting the frozen egg bonus the freeze timer correctly appears and all enemies remain stationary, once the freeze timer was 0, it disappeared and all enemies (hunter, wolf, bat) began moving normally again.

Speed bonus test:

Upon collecting the speed bonus (fire egg) a speed timer appears on the screen, the user speed is increased but is still able to die to enemies and still unable to move through walls. Upon colliding with a trap the user's speed bonus is removed abruptly even if there was time remaining on the bonus.

Score bonus test:

Upon collecting the score bonus (golden egg) the user's score correctly updates, increasing by a random integer up to 5.

Winning the game test:

Finally, we tested the door (portal) while there were still eggs on the screen. It was tested that the portal acted as a wall in the sense that users could not pass through it. Once all the regular eggs were collected from the screen the portal animation updated and upon touching the portal the game ended and the win screen was shown successfully.

V. Conclusion

During development we discovered that IDE's sometimes couldn't read from the path for the original load image function. In this phase, we updated the load image function to be path independent by using the getResourcesAsStream function to allow the image loading function to work seamlessly for the command line and IDE. During this phase of the project, we also performed some beta tests with friends and family in order to receive feedback. The main feedback received was to update the hitbox of all objects (the collision function previously treated objects as set cell sizes so the user would collide with another game object before the pixels of the character touched). The collision function was then updated to improve the hitbox.

One of the largest challenges we faced was testing our maze class. Our maze class performed all of our game's logic and relied on user input. By creating a maze it began the game which runs automatically calling many functions which were all coupled. This made making a unit test that isolated single functions or tested specific implementation impossible so we had to perform black-box functional tests instead of structural tests. Structural tests would have been valuable to confirm correct algorithm outputs for example with the checkCollisions function. Since all the characters and variables within the class were also private we could not assert much behaviour while running our black-box tests. Looking back we would have modified our design and split up the maze class into multiple smaller classes (give checkCollisions method to character class, make a map specific class, make a gameLogic and GUI class, make a Bonus superclass), this would make the individual classes and functions less coupled and far more testable through assertions. We could have created public objects to allow for assertions but we did not want to sacrifice encapsulation for testability.

While we were able to perform many structural-based unit tests for most of our individual classes such as characters, and environment objects to assert the proper behaviour according to function-specific implementation, the bulk of our game's logic and functionality was only able to be black-box tested according to our specifications. We learned firsthand that compromising good design and low coupling for convenience by making a large class can cost you when it comes the time for testing. No major bugs were found during this testing phase which could be in part due to the rigorous functional testing we were doing throughout the entirety of phase 2. After running all of our unit tests and integration test, the code coverage was 90% and the branch coverage was 85%. We deem the test case coverage satisfactory as the leftover coverage can be tested by manual testing.