Simon Fraser University
Faculty of Applied Science
CMPT 276 - Introduction to Software Engineering

Term Project - Phase 3

Testing Phase Report

Group 5

Overall Approach in Testing Phase

Our group splitted Testing Phase into 2 main stages. The first stage was physical visual tests followed by the latter stage which is JUnit tests. We adopted this approach as we acknowledged the fact that we would not be able to examine all parts of the code if we were to only use one testing method for the entirety of this phase.

Stage 1: Physical Visual Test

The first stage was conducted primarily to test Graphical User Interface (GUI) of the game. We realized that this would be the best way to test the GUI as there is no available tool in JUnit to test it directly. In this stage, all group members were tasked to play the game multiple times and interact with all the buttons in the game such as pause, mute and control. Everyone will evaluate and raise any suggestions for improvements or bugs when playing the game.

Stage 2: JUnit Test

The second stage consisted of implementing JUnit unit tests that checks for logical and semantics errors in the code. In this stage, we would write JUnit test cases for small code snippets of the methods in the classes.

Measures Taken to Ensure Quality of Our Tests

Each of us were responsible for JUnit unit tests on all the classes that we were tasked to implement during the implementation phase respectively as he/she will have the most knowledge on how that class should be tested effectively. During the initial period of this phase, we discussed and added the main functionality that needed to be tested into our to do list in GitLab. This ensures that we will not miss out any critical functionality of the game. In addition to that, we also applied some additional measures to maintain the standards of our unit tests.

Measure 1: Structural Testing (Line Coverage)

Our group focused more on the structural-based testing. For each and every class, our unit tests take into account every method of each class. Each unit test will cover all specific amounts of code in the production code to ensure that every line of code is being evaluated by one of the unit tests. We partitioned production code using the unit testing technique based by their functionality and had at least one of the unit tests evaluating the code. As an example, we will look at one of the unit tests we had which is ExitBoardPathwayTest() in generatorTest. That function tests the code snippets which randomize the exit's position in the map to ensure there is actually a path for the player to reach the exit using Breadth First Search Algorithm. That function provides line coverage for parts of 2 classes which are Level Generator and Location Randomizer Generator class.

Measure 2: Structural testing (Branch and Path Coverage)
Many of the unit tests we had, tried to cover the method's path as much as possible. Since our game encompasses 3 different levels, our program's main branching revolves around these 3 levels. This is due to the fact that each difficulty will produce different amounts of walls and enemies. Therefore, we parameterized the 3 different levels into our test to appraise the behaviour of the function as a way to expand our branch and path coverage for each function.

Measure 3: Construct Expected Result Table for Tests that Requires Combination
In order to take in account all possible results, we created a table for those parameterized tests before running the test. The table documents all the possible combinations and expected results from those combinations. It ensures that we would not miss out any critical combinations. Table 1 shows an example for testValidMove().

Tests No.	Position being Passed Into Function	Original Position In the Function	Expected Result
1	(5,2)	(5, 3)	Passed
2	(5,4)	(5, 3)	Passed
3	(5,5)	(5, 3)	Failed

Table 1: An example of Expected Result Table for testValidMove() in EnemyTest.java

Code Coverage
As mentioned above, we took measures in our unit tests in order to attain as much coverage as possible. However, we approximately attained 70% coverage for the gameLogic package while only achieving approximately 10% coverage for the UI package. Table 2 shows all the classes we were not able to test through JUnit. We were not able to achieve full coverage of the gameLogic package due to design limitations of the class. Some methods were either private, required dependencies from other classes or impossible to test due to the limitation JUnit. For example, we were not able to fully test the playSound() functionality of the Music Class since JUnit does not have any assertion for that criteria. Therefore, we were only able to test the method in general so we do not count that as actually achieving coverage on that part of the code. In a different context, for the UI package, the reason for such low coverage on UI is mainly

also due to the limitation of JUnit. Therefore, we looked into multiple solutions into testing UI, such as adopting Selenium. However, due to time constraints, we were only able to conduct physical tests on the user interface.

Our unit tests do acquire a high branch coverage as they all take in account of the three main branches in our program which are the levels of difficulty. As mentioned before, when applicable, we parameterized all test methods with the three different levels to ensure all functionality works as expected for all three main branches.

Classes Not Being Tested By JUnit
Button.java
ControlScreen.java
DrawDead.java
DrawLive.java
EndScreen.java
Gamescreen.java
Interface.java
Main.java
Misc.java
newFont.java
NextScreen.java
PauseScreen.java
Sprite.java
TitleScreen.java

Figure 2: Table shows classes in the game that could not be tested using JUnit.

Findings

Our findings will be based on our own internal code review session, results from both stages in the testing phase and suggestions from our professor and teaching assistant (TA). For our findings, we mainly discovered some inefficiencies in our code designs and some unobvious bugs.

Findings 1: Unobvious Bugs Being Uncovered Through Repeated Testing

Since our game depends on randomization for many of its components such as location randomization of objects, we were not sure on the consistency of our program. Through testing, we were able to test a component multiple times to simulate different scenarios and found bugs that were not obvious at first glance. For example, when we were running ExitBoardPathwayTest, we realized that the test will fail occasionally. Upon further inspection, we discovered that a missing condition has caused the exit to randomly spawn into the walls of the maze. Furthermore, physical tests did make some bugs float to the surface. For instance, we observed some minor bugs in the user interface of the game such as two enemies standing on the same position and a “blue” ring appearing on the pause button when it is being pressed.

To sum this finding up, we were able to fix all the bugs that were uncovered through this repeated testing method.

Findings 2: Inefficiencies of the Initial Enemy’s Chasing Algorithm

In addition to those bugs, we identified a design flaw in the enemy’s chasing algorithm. Our old algorithms do not account for every available case. One of the cases that it would not account for was that it will be stuck in an up down motion endlessly when it is at a specific orientation and at the corner of a three-faced wall. Furthermore, as more physical testing is done on the enemy’s chasing algorithm, more and more bugs were uncovered on this algorithm. In the end, we concluded that it is much better to rebuild the algorithm from ground up as fixing the initial algorithm will result in a huge function with a lot of helper functions. Furthermore, it adopts some of the bad coding designs such as hard-coded values which results in difficulties in future refactorization.

The new algorithm uses Breadth First Search (BFS) to search for the shortest path to the player from the current position. The reason we chose Breadth First Search as our new algorithm is that the algorithm is able to account for all the different demographics of the maze and it is able to do so efficiently. Furthermore, due to the reason that I will mention in the next findings, we also decided to split out this algorithm from board class and made it a seperate class called PlayerFinder.

Findings 3: Flaws in the Design of Board Class

During the implementation of the unit tests, one of our group members uncovered a major flaw in our design. The board class was essentially a "blob" class and it caused some difficulties testing the code in other classes due to their dependencies to the board class. With these dependencies, it was impossible to test the code of other classes independently. Furthermore, the high complexity of the board class requires the application of complex and indirects tests to test the class. It indirectly hampers the effectiveness of the test.

In addition to that, the board also shows signs of "Copy and Paste" programming which is a bad design. In elaboration, the board class maintains two attributes which are Traps and Rewards that have similar structure. As a result of this, the board comprised similar codes for Traps and Rewards. One clear example was isTrap() and isReward() which have the same implementation approach.

This issue was raised to the group immediately. After much discussion, we decided to completely refactor the part of board class into multiple separate classes which are BoardArrayManager, LocationRandomizerGenerator, ArrayFinderManager, RewardArray and TrapArrayClass. Furthermore, ArrayFinderManager also applied the factory method design as a way to counter the "Copy and Paste" programming issue mentioned earlier. RewardArray and TrapArray classes extend separately from ArrayFinderManager. With this new design, we were able to achieve higher data encapsulation and better modularity.

Conclusion From Findings.

From all these experiences and findings, we realized the importances of the testing phase. Tests done during the testing phase are able to dig out bugs from places we less anticipated and blemishes in the design of our program. Furthermore, we also acknowledge the importances of good class designs such as modular code as it prevents our code veering into bad code designs such as spaghetti code and blob class. Prevention is better than cure.