Implementing Test Cases

Our group started out by listing and categorizing our games main capabilities and functions which would be tested. These categories included user interface, collision detection, sound effects and music, map generation and input handling among others. An example of features under map generation were the placement of tiles, objects and entities onto a specific game map and switching between different game levels. Once this was completed, we were able to write specific unit and integration tests that would evaluate each one of these game features.

To simplify the process of organizing our test cases, our group tried to make our test classes as cohesive as possible. For example, test methods involving our Player classes would be written in the PlayerTest class. Features involving the use of multiple classes would be tested in whichever class would have the most change.

During this testing process, our team learned that making unit tests for games is a lot more difficult than expected. As our game is written using object oriented patterns, we couldn't test our game with a functional programming approach. Many of our methods also did not take any parameters and only performed operations on its own attributes. Thus, instead of testing out different input values, we designed unit tests that tested our game's features based on specific class attributes like player health.

Since many of our classes contained void methods and private attributes, writing tests to verify whether they functioned correctly was a challenge. A palliative workaround we utilized heavily involved creating helper subclasses that extend the target class in order to override or implement methods to help us write our test cases. This strategy also helped us create tests for abstract classes, reducing the number of tests we had to develop since we would only have to write one set of tests for the methods in the superclass rather than writing identical tests in every subclass.

In the end, our team achieved roughly 80% code coverage with our unit and integration tests which included a majority of our games featureset.

Package Name	Code Coverage (%)
App	92.6
Audio	91.9
Entity	95.6
Input	46.2
Map	73.1
Object	94.2
Pathfinding	100.0
Settings	86.2
Tile	96.1
UI	52.8
Total	80.4

A portion of our codebase that we were unable to test automatically were features related to our user interface and input management. Luckily for us, these do not need to be heavily tested using unit or integration tests as any bugs or defects concerning them can be discovered by inspection. For example, if the user interface displays an image incorrectly, this can be very easily detected during playtests as we can see the interface on the screen and tell whether it is producing the correct images. Flaws with user input can also be identified quickly this way.

Therefore, we are confident that these features will be functioning correctly as we have playtested our game extensively.

One of the biggest challenges we had was writing tests related to collisions between objects and entities. This was due to the fact that many of our classes relied on each other to function correctly. Consequently, in order to test collisions, we had to create an instance of our entire game inside our test class and create test methods that simulated collision events. For instance, if the player's health is not full, collecting eggs will replenish their health; otherwise, their score will increase by 100 points. In order to test the logic for this feature, we would instantiate the game, set the players health, teleport the player to the nearest egg, collide the player with the egg and check the player's score and health.

Bug Fixes

Upon running our unit and integration tests, our team managed to discover quite a few bugs and defects in our codebase. A major problem that we uncovered was that the coordinates of the player's hitbox would increase perpetually if no farmers were loaded onto the map, resulting in the game crashing as the hitbox would be indexing invalid positions in the map array. This proved to be a significant issue as we were unable to play test levels without farmers. However, the fix for this bug was relatively simple and involved making sure the player's hitbox location was reset during collision checking even if no farmers were present on the map.

It was also possible for the pathfinding algorithm to attempt to create a path to the player that would result in the farmer exiting the map and causing an out of bounds error. This was never encountered in regular gameplay but integration testing revealed that it was possible under the right circumstances.

A number of smaller bugs were also found and fixed. This included being able to make a “clucking” sound when using chicken’s cluck ability even if it was still on cooldown; this meant that farmers wouldn’t be frozen which could cause some confusion for players..

Changes to our Code

While designing our tests, we realized that a large portion of our codebase was simply not written well and would not be easily tested. Many of our classes violated common software engineering practices, one of the most egregious being our codebases’ high coupling between classes and low cohesion for instance. To remedy this predicament, a number of holistic changes were made to our game.

Our group broke up many of our larger classes into several smaller ones such as our InputHandler and UI classes which contained upwards of 700 lines of code each. These classes contained different functionality depending on the game’s current state; therefore they were split up into several game state specific subclasses which inherit an abstract class containing their shared attributes and methods. Then, a so-called “manager” class would then tie these subclasses together along with the logic required to move between the different game states.

Since most of our classes relied on other objects instantiated within the GamePanel class, our team decided to update several of them to use the singleton design pattern. This included our games Settings, TileManager and Audio classes since most of these did not depend on other classes.

Another change included creating new classes to increase the cohesiveness and organization of our codebase, such as our StateManager class. This class in particular proved useful as it allowed us to more easily switch between different game states in our game logic and in our integration tests.

Alongside our automated JUnit tests, we also conducted numerous informal play tests to evaluate the gameplay of our game which led to several modifications and improvements. Early playtesting had shown that our level designs were too challenging as the player could be stuck between objects and farmers with no way to escape. This would lead to a great deal of frustration among our group mates while they played the game. Therefore, we added a new ability that allows the user to freeze the farmers around them in a five tile radius in front of the player. A cooldown was also eventually added as the player would be able to spam the ability to effectively freeze the farmers permanently, making the game too easy.

We also increased the time before an egg disappeared or “despawned” to 30-90 seconds instead of 30-50. This change was made as our levels ended up being larger than we expected during our development phase, so several eggs on the map would despawn before the player would have a chance to collect them. This is an example of a flaw that wouldn’t have been discovered by purely using automated testing.