Identifying features and classes to test:

The format of our testing first involved the planning and overlook of where and what we should test initially. Our first meeting was the assessment of our classes, with each member contributing to what they think they would be able to individually test based off of the classes they created. After this initial step, we determined we had several classes that could be tested solely by themselves and divided up the work. The classes we initially came up where mainly the objects with methods that we could test (like Squirrel, Raccoon, Orange, Potato, etc.), since we could use the assert method to compare, however as we discussed and researched further our testing domain grew into more and more interactions and dependencies on other classes.

The next step in our group was looking at some of the more challenging testing (still functional testing), with classes that seemed like they were unable to test. After group discussions we found ways to test these classes effectively and efficiently. These certain tests we knew would be more integrated with other classes, so we had to collaborate on how we could figure out to get a test result that would be comparable.

Important interactions

As discussed above, in our initial process of determining testable features and classes, we also noted important interactions. In our game we implemented our game with certain classes handling certain features, and other classes handling other parts of the game. An example of this would be the Sound or UI class, handling their features, but then are used in bigger classes like GamePanel. Our classes that combine most of our classes are GamePanel, KeyHandler, and __ handle all the information going to the game making it easier to

Measures for Quality testing

Code Coverage:

During the process, we prioritized the interactions over metrics like line/block coverage. We believed that this was the best approach to see if our game was functioning as intended. However, we also made use of the block and line coverage metrics to evaluate the quality of our tests. We were able to achieve a higher coverage rate as we covered more interactions. Below is the summary of our code coverage for different classes of our project.

> KeyHandler.java: The coverage measured was 83% (200/242). The segment that was not covered was System.Exit() and the method called KeyReleased. System.exit() wasn't tested. KeyReleased function does not have to be tested since the test methods in the test file declare the key press by integers and once it is used, it automatically releases.
>
> Setter.java: The coverage measured was 100% (373/373)
>
> Items.java: The coverage measured was 100% (21/21)

PathFinder.java: The coverage measured was 99% (513/516)

Sound.java: The coverage measured was 97% (86/89)

Potato.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

Acorn.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

Orange.java: The coverage measured was 47.6% (20/42). draw() method was not tested since it only draws on the screen. And catch block since it does not throw an exception

GamePanel.java: The coverage measured was 40.9% (138/337). The lines of code that are related to Threads were not tested. Also, the method that is related to draw() is not tested.

UI.java: The coverage measured was 6.9% (50/728). The position of contexts and images are not tested.
When it comes to adding unit tests, it can be challenging to do so for classes that are more graphics-related. This is because graphics-related classes often have a visual output that is not easily testable with traditional unit testing methods which is why some classes were left out of unit-testing.

Tile.java: The coverage measured 100%.

TileRepresenter: The coverage measured 56.5%(190/336). The draw method in the TileRepresenter class is not tested since it draws on the screen and does not return or change any other data to test.

Squirrels.java: the method coverage was 77%. didn't test for the draw() part and some default parts. draw() just draw the images.

Raccoons.java: the method coverage was 90%, didn't test for the draw() part and some methods were updated. the draw() just draw the images. And raccoons were set with the static initial values from Setter, so some of the code just initialized the value for those methods created in other classes, like pathFinder.

Portal.java: The coverage was at 54%. The draw() method was not tested as well as the exceptions.

Stopwatch.java: The coverage was at 45%.There were some cases that were not tested as they were duplicated tests and were redundant.

Findings

Through our testing we discovered a couple of different things But overall the testing phase of our project revealed little in terms of coding errors. The first thing we noticed was the interactions between classes and functions of our project. We discovered that the attributes that allowed access to the variables were hindering the accessibility of the variables in our testing function.

1. We have learned that writing efficient and effective tests takes work. Finding the interaction between classes and methods was quite complicated due to coupling.
2. More than half of our classes are connected to the GamePanel class, and refactoring would involve a considerable rewrite.
3. During Phase 3, we made little or no changes to our program code. We did not face any bugs in the class files.
4. However, as mentioned in Finding 1, due to how we implemented our project, high coupling style code, it was very complicated to create perfect test suites for all the classes and methods. It might be why some of our code coverage is not high enough.

Breakdown of our Important Tests

Test file name: KeyHandlerTest.java

Interaction between Keyboard input and UI design -- Title Page

Whenever the game has launched, the tile page shows up first then the user chooses the next step by pressing the keyboard (up, down, left, right). The title page (UI design) interacts with the keyboard input. There are two options which are “NEW GAME” and “QUIT” to choose from the user. By pressing the up arrow key and the down arrow key, the user can choose either one. Therefore, it is important to test whether pressing those two keys properly navigates on the title page.

Test method name: titlePageInteractionTest()

Interaction between Keyboard input and UI design -- In-Game

While in the game, the user can press either ‘p’ or ESC from the keyboard to pause the game. This has to be tested to determine whether those two keys are properly working to pause the game. Furthermore, when the user presses those two keys one more time, the game has to resume.

Test method name: inputPauseTest()

Interaction between Keyboard input and UI design -- Game-Over

After the user dies or accomplishes the tasks in the game, the screen changes to the Game Over UI screen. On that screen, the user has to choose either “RETRY” or “QUIT” by pressing the keyboard (up and down). It has to be tested to determine whether pressing those two keys properly navigates on the options on the Game Over page.

Test method name: inputGameOverTest()

Conclusion:

Throughout phase 3 of finding testable features, configuring the functional tests and structural tests, and reviewing, we didn't find many errors. In phase 2, when we were trying to run our game we did a full code review session in order to get our game running, where we corrected most of our errors and logic of the game. In the test phase, we only found a handful of errors that were syntax errors (which we found since the test wouldn't run), other than that it was just looking at the game from a structural point-of-view to see how we could have refactored it differently. Overall the testing was successful and proved the quality of the code written.