Phase 3 Report:

Logic-Based Test Cases:

EnemiesTest:

EnemiesTest in general covers the Enemies' class general logic. We were able to determine how separate variables interacted, and we checked if the initial values were initialized to their proper values. We also checked if movement worked as desired.

BonusTest:

BonusTest covers the Hostage class's general logic, given that all generated tiles are subclasses of the Person superclass. Health was tested to see if it's health variable decreased properly, and made sure it was set at 1. Damage, speed, movement and damage taken was checked to make sure Bonus is treated like a bonus item.

GoalTest:

GoalTest covers Goal classes general logic similar to bonus tests. Additionally we were able to determine if the goal variable was initialized to the proper value (1, which is extremely important as we only want a single end goal tile collision), and we tested a decrement of Goal..

HostageTest:

HostageTest covers the Hostage class's general logic, given that all generated tiles are subclasses of the Person superclass similar to previous item-like classes

PlayerTest:

PlayerTest covers the Player class's general logic, given that all generated tiles are subclasses of the Person superclass, the test methods in Player asserted the constructor variables were declared properly. For example, Player Health must be equal to the PLAYER_HEALTH variable in Settings.java, likewise with Player Speed to PLAYER_SPEED. Additionally, we asserted the Player damage was set at 1, which was important for future testing.

ScoreTest:

Arguably one of the most **important** classes, the tests in Score test mainly elaborated on the hostageCount integer, it was incremented several times to see if it incremented properly. Without hostageCount working properly, players are unable to reach a win state, so it was vital to make sure the methods in Score worked. We used a for loop to increment hostageCount to 4, and used the goal() function in Score as a test case. If hostageCount is at a value of >=4, then it will return True, else it returns False. The goal() function in Score.java is used to reach the end state in our program, and it was important to ensure all the tests ran consistently and smoothly.

We also tested boolean variables in Score.java to make sure they worked as well. Another important aspect was creating test cases on score counts, and making sure the possible scores are within intended ranges. The two variables that truly affect the score outcomes are whether the player wins and/or has the bonus. So four tests cases were made to assure that scores are within those ranges depending on the two conditions.

SettingsTest:

SettingsTest tested the global variables that were used throughout the entire program, consistenting PLAYER_HEALTH, PLAYER_SPEED, SCENE_WIDTH, and SCENE_HEIGHT. We did tests to make sure the values matched the values stated in the Settings.java file, and all were found to be conclusively true.

Integration Test Cases:

In regards to integration tests, we mainly aspired to see how class specific integration worked with each other. For our group, basic aspects of the interface were far better adjusted through manual testing. As an example, classes with runtime execution were better suited to vigorous testing through as many inputs as possible. Interactions such as function win scenarios could only be tested on runtime execution, and were better suited as so.

Thus, our integration testing had a heavy focus on **cross-class variable interaction**. We already know for a fact from the previous tests, the general game code is sound for the individual classes. Therefore the next step was to see how code interacted throughout the program, not just across superclass to subclass (parent-child class) relations.

We came up with important class interactions that had to be thought up off. The first test was to see the general interaction between Player health and Enemy damage, we tested if logically three ticks of enemy damage (at one base damage) was equal to the total number of hearts a player had, three. Several important interactions such as these were repeated to test the general logic of the game across classes. We checked for issues with interactivity in-between the Controller class, and it's surrounding classes. The Controller class handled the object generation of the game, and thus we appropriately tested the cross-class functionality of said class.

The main test however we focused on was **Character Collision**, it's instrumental to the running of Rescuer, and functionality everything depends on CharacterCollision. We implemented a test that showcases character collision interactions between two Person-type objects, and proved the functionality of the bulk of our code. We can see Character Collision occurring frequently throughout runtime, but it was good to get affirmation of what we already knew, for extra assurance. Most of this testing was actually done in **PlayerTest.java,** you can see the results of the collisions there.

Measures taken for ensuring the quality of your test cases:

For the most part, the possible errors/bugs that could be found in the code were **logic related**. Thus, we initialized variables that matched the ones found in our code, and tested the logical equivalence of them. Several classes were tested in the /src/test/java/ folder, for important tests to ensure none of our functions were overcounting, that our variables were initialized properly, etc.

It was of the utmost importance to iterate through the logical variables of each class, it gave us the confidence that the game (truly) functioned properly and had no possible failures. As an example, in our Score class, our hostageCount int variable is initialized to zero. So we assert that the hostageCount is equal to zero, and test for specific related functions in Score. The function increaseScore was tested and found to be an efficient incrementer for the hostageCount variable. If there was a possible way for goal() to return early, then we would have a game-breaking bug in the code, so as reiterated the tests added a bit of validity.

As mentioned in lecture, it's obvious that even despite all the logic tests done, it is still possible for some miscellaneous bugs to appear in the game code. However, after the final finished Phase3's logic tests, we did find an **absence of logic bugs**, and acknowledge that there is still a slight element of uncertainty, no amount of testing can give a programmer 100% confidence that there are zero bugs. We are confident that the testing done covered as much of the classes as possible, and that we appropriately covered line & branch coverage thoroughly throughout our testing.

Findings:

One of the findings occurred later on in development and concerned ending scores. When creating tests for scores I realized that if the player took over five minutes to complete the game, the ending score would be negative since our scores were time based. This means the ending score is multiplied by the upper bound of 300 second minus the time played. A simple and effective solution was to simply take the max value of zero and possible score, so if the player gets something less than 0 it would default to a score of zero. While we did not find many logic-based bugs, we did use the findings as ways to fix any extraneous outliers in our code. In general, this phase was a huge learning opportunity to **improve** and polish on our production code. We revealed and fixed any extraneous logic bugs and improved on the general quality of our code.

This Phase was extremely important for working on the general flaws & imperfections. We learned to divide tests into subsections and then into smaller parts which became our unit tests. We also had to make compromises which ultimately lead to hours of rigorous manual testing for our user interface components. and we found the test cases portion to be vital for this level of improvisation.