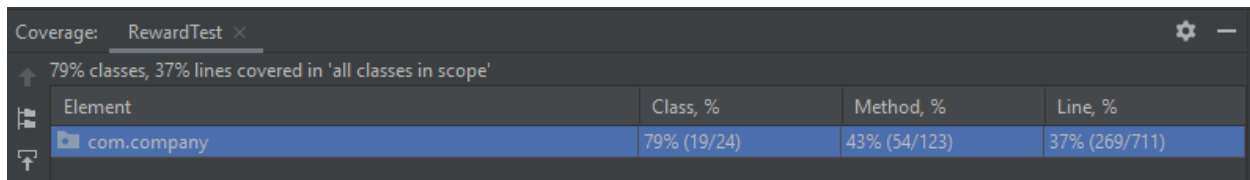


Group 6: Phase 3 Report

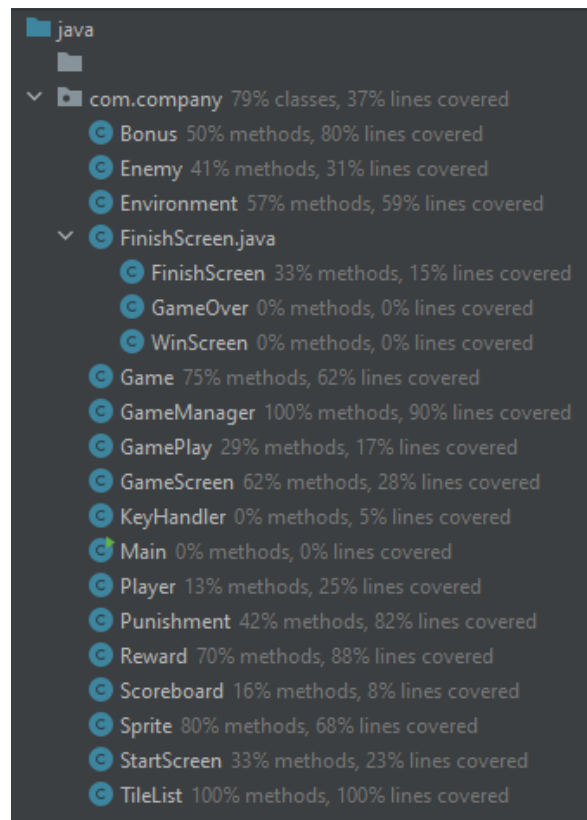
Various JUnit tests were written in the test folder of the project's hierarchy throughout Phase 3 in order to comprehensively test the previously implemented features from Phase 2 for the overarching idea of Zombie Dash. As a group, we decided that the features that had to be tested included any aspect of the game that was actively running and called on, however, not all features were tested. While line coverage would have been a simple metric to judge the comprehensivity of our JUnit tests—and seemed like a suitable measure to start with—it was also true that line coverage had its own flaws. Figures 1 and 2 depict work-in-progress screenshots pertaining to coverage as determined by the IntelliJ IDE.



The screenshot shows the IntelliJ Coverage tool window for a test named 'RewardTest'. It displays a table with coverage data for the 'com.company' package. The table has four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The 'com.company' package is highlighted, showing 79% class coverage (19/24), 43% method coverage (54/123), and 37% line coverage (269/711). Above the table, a summary bar indicates '79% classes, 37% lines covered in 'all classes in scope''.

Element	Class, %	Method, %	Line, %
com.company	79% (19/24)	43% (54/123)	37% (269/711)

Figure 1. Screenshot from IntelliJ depicting the class, method, and line coverages of a work-in-progress JUnit test for the Reward class. Disclaimer: The screenshot may not be representative of the final JUnit test, but the line coverage of JUnit tests at the time of the photo was insufficient in terms of solely measuring the number of lines covered. A 269/711, or 37% coverage score would appear fairly inadequate at a cursory glance, but would be later modified during our testing.



The screenshot shows the IntelliJ Project Explorer with a tree view of the project structure. The 'com.company' package is expanded, showing a list of classes and their coverage percentages. Each class is preceded by a circular icon indicating its coverage status. The classes and their coverage are: Bonus (50% methods, 80% lines covered), Enemy (41% methods, 31% lines covered), Environment (57% methods, 59% lines covered), FinishScreen.java (33% methods, 15% lines covered), GameOver (0% methods, 0% lines covered), WinScreen (0% methods, 0% lines covered), Game (75% methods, 62% lines covered), GameManager (100% methods, 90% lines covered), Gameplay (29% methods, 17% lines covered), GameScreen (62% methods, 28% lines covered), KeyHandler (0% methods, 5% lines covered), Main (0% methods, 0% lines covered), Player (13% methods, 25% lines covered), Punishment (42% methods, 82% lines covered), Reward (70% methods, 88% lines covered), Scoreboard (16% methods, 8% lines covered), Sprite (80% methods, 68% lines covered), StartScreen (33% methods, 23% lines covered), and TileList (100% methods, 100% lines covered).

Class	Methods, %	Lines, %
Bonus	50%	80%
Enemy	41%	31%
Environment	57%	59%
FinishScreen.java	33%	15%
GameOver	0%	0%
WinScreen	0%	0%
Game	75%	62%
GameManager	100%	90%
Gameplay	29%	17%
GameScreen	62%	28%
KeyHandler	0%	5%
Main	0%	0%
Player	13%	25%
Punishment	42%	82%
Reward	70%	88%
Scoreboard	16%	8%
Sprite	80%	68%
StartScreen	33%	23%
TileList	100%	100%

Figure 2. Screenshot from IntelliJ depicting the coverage of methods.

We considered how best to approach the judgements about whether it would be acceptable to test certain sections of code. For instance, some lines of code used for spacing or bracketing purposes, did not contain any meaningful content to test. Additionally, getters, setters, and private attributes were not justifiable as JUnit tests, either. Private attributes should have remained private, and testing getters and setters would have been redundant and a poor choice of function to execute given the more elaborate functions to exist. Additionally, the getters and setters in the first place were created with the intention of being called in other functions, thus, separate tests would not have been warranted since the coverage may have already been taken care of in other classes.

Although this is not an all-inclusive list, some of the isolated features and functions we tested during this phase, sorted by classes, are outlined below with the according justifications for their testing:

- *Enemy.java:
 - randPosition:
 - Function: Sets the position randomly.
 - Test: Call the randPosition function to obtain the location of the objects. Assert that PosX and PosY are greater than zero, but smaller than the maximum position values implied by the walls of the map. Ensure that the resulting position is valid on the map before returning the corresponding value.
- ScoreBoard.java:
 - addScore(Player player):
 - Function: Adds the player's name alongside their score to the arraylist ScoreBoard.
 - Test: Verify that the scoreboard output contains the updated score for the player.
- Gameplay.java:
 - checkFinish():
 - Function: Checks whether the game is winnable depending on 1. Whether the game has started yet (i.e., not on the start menu), 2. The Player's location (i.e., cannot win a game without reaching the exit), and 3. Whether all the rewards have been collected (i.e., prerequisite for winning the game, alongside the other conditions)
 - Test: Application of condition coverage to determine the conditions met by the current state of the game in varying branches for different situations. Verify if the aforementioned attributes in the function correspond to the winning state of the game.
 - checkNearbyEnemy():

- Function: Compares the distance between an Enemy and the Player. Returns true if the Enemy is within a certain threshold of the Player, false otherwise.
 - Test: Assert that the function returns true if the position of the Enemy relative to the Player is within range. Assert that the function returns false if the position of the Enemy of the Player is outside of the acceptable range.
- isAllRewardCollected():
 - Function: Compares the number of normal rewards that have spawned at the start of the game to the current number of rewards that the Player has collected. Returns true if the two values are the same, returns false otherwise.
 - Test: Assert that the function returns true only when the number of rewards picked up by the Player is equivalent to the number of normal rewards (i.e., excluding bonus) spawned in the game.
- *Player.java:
 - winScore():
 - Function: Increments the Player's score by one.
 - Test: Determine whether the Enemy object is located at the specified x-coordinate as intended upon calling this function.
 - loseScore():
 - Function: Decrements the Player's score by one.
 - Test: Determine whether the Enemy object is located at the specified y-coordinate as intended upon calling this function.
 - resetScore():
 - Function: Resets the Player's score to zero when initializing the game or reaching a victory/game-over result screen.
 - Test: Assert that the Player's score is actually being assigned the proper int of 0.
 - winReward():
 - Function: Increments the Player's score by one upon obtaining a normal reward.
 - Test: Assert that the Player's current score is one greater than its previous score prior to obtaining the reward.
 - winBonus():
 - Function: Increments the Player's score by ten upon obtaining a Bonus type of reward.
 - Test: Assert that the Player's current score is ten greater than its previous score prior to obtaining the Bonus.

*All tests were later adapted to an overarching GameObject class in the refactoring process described later in this report.

The different user interfaces of our system—namely the StartScreen, FinishScreen, and GameScreen classes—appeared to seamlessly interact with the rest of the components. The only file system that was in our project was relevant to the resources folder to extract image files for the Sprite class. The interactions between the front-end, graphic representations of Zombie Dash, as described in the aforementioned classes, were effective enough in conjunction with the code of the game’s mechanisms layered underneath every Sprite movement. Integration tests for the interface classes were accordingly conducted, although understandably not completely isolated due to the nature of interface design coordinating with other classes to produce the game output. However, refactoring was done for the Game class by moving all interface/GUI related methods and variables to its own class called ‘Interface’.

In order to guarantee the quality of our test cases, we followed the Modified Condition/Decision Coverage (MC/DC) approach to find the most efficient combination of test cases, as per path coverage rules. For test cases with many conditionals, determining on-points, off-points, in-points, and out-points were crucial to finding the most relevant JUnit testing conditions, as well as avoiding redundant tests by only focusing on the combinations of conditions that produce change in state.

While several bugs on a smaller scale were fixed, one of the most notable adjustments made to our code included resolving an occasional bug in which the incrementing and decrementing scores of several object types (e.g., Rewards, Bonuses, Punishments) were applied to the Player before the start of the game, as the objects may have been at the same position prior to the map being drawn when the game actually started. As the Player would have detected the nearby Rewards, Bonuses, and Punishments due to their close proximity to start, the Player would have gained and lost a substantial number of points without genuinely having played the game yet. This bug was later fixed by adding a boolean feature to the relevant Java class to signify whether the game had started or not, and that the objects could not have been collected or traversed through before beginning the game. Thus, this incremental and decremental score issue was fixed through the JUnit testing process.

Nonetheless, we learned much about the testing phase by actually writing and conducting JUnit tests ourselves. In addition, we thought of better ideas to format and refactor our code as a result of the testing we conducted, and performed reviews of our code as deemed necessary. In fact, the refactoring process warranted moving around our previously established tests, and we further adapted the functions to an overarching GameObject rather than separate classes for every game entity. This was accomplished by making the classes Player, Enemy, Bonus, Reward, and Punishment to be subclasses of GameObject instead. This change was spurred because some functions were shared between all of the aforementioned entities—such as checking for valid movements and setting x and y-coordinates, for example—and were made more concise from refactoring. These changes were denoted by [Old Test] in the previous pages of this report. By making these game objects as subclasses of GameObject, the code for each class was made much cleaner and less redundant. Three other methods which implemented the same functionality, such as initializing and randomizing the reward, punishment, and bonuses were also refactored to one single method called ‘setRandObject’

which takes in any ArrayList of a subclass of GameObject rather than repetitive and redundant methods for each separate class object. Similar refactoring was also done on previous methods 'checkNearbyX()' which updates to read if the player's position hits a certain object's position. Four CheckNearby methods were refactored and replaced by one refactored method called 'checkObject()', which takes in any GameObject subclass as its parameter and implements the appropriate in-game response by running 'resultForObject()'.

As described above, the GameObject class condensed the previous JUnit tests as per the refactoring process:

- GameObject.java:
 - setPosX():
 - Function: Sets the x-coordinate of the object's location.
 - Test: Set the x-coordinate to a value and assert that getX() is equal to the value.
 - setPosY():
 - Function: Sets the y-coordinate of the object's location.
 - Test: Set the y-coordinate to a value and assert that getY() is equal to the value.
 - ValidMove():
 - Function: By checking the attributes and position of the surrounding tiles of the map, this function verifies whether the intended tile to move into is traversable (i.e., not a barrier or wall).
 - Test: Use the map to set a specified location as a barrier, before asserting that it returns false. Similarly, set the location to a road, and assert that it returns true.
 - randPosition:
 - Function: Sets the position of the GameObject randomly.
 - Test: Calls the randPosition function to obtain a randomized position for the GameObject. Assert that the PosX and PosY of the object are greater than zero, but smaller than the maximum position values implied by the walls of the map. Ensure that the resulting position is valid on the map before returning the corresponding value.

Thus, the JUnit tests also became more concise in the final version of our Phase 3 submission and evolved over time as well into more presentable and readable code. The hands-on work of this testing phase solidified our understanding of the software development process, and modifications to our code were accordingly made during the testing process to account for any exceptions that may not have been noticeable from the first implementation of Zombie Dash from Phase 2.