Coverage of tests:

- Coverage of tests:
    - ObjectSetterTest:
        - 100% of classes covered (3/3)
        - 10% of lines covered (18/179)
    - GameStateTest:
        - 100% of classes (5/5)
        - 10% of lines covered (26/266)
    - PlayerTest:
        - 19% of lines covered (64/334)
        - 30% of methods covered (7/23)
- Branch coverage of tests:
    - Player movement has four branches from 4 different outcomes: user inputs ‘up’, ‘down’, ‘right’, or ‘left.’ All four of these outcomes were tested in our PlayerTest. The branch coverage in this case is 4/4 = 100%.
    - GameStates had 7 branches consisting of the main states: Playing, Pause, Win, Lose, Restart, Exit, and Menu. These states were tested under the file known as GameStateTest. Branch coverage is 7/7 = 100%.

Quality measures taken for each Test

GameState Test:

For the GameState class, we concluded that there are three main components to check. The first was that Game states were successfully rendered, and that the states successfully switched according to preconditions. The next was to determine if the variables associated with each state are true (eg. GAMEOVER, meant score was negative, or moving enemy collision occurred is true). Finally, the verification of boolean expressions of the states were correct and that there was no overlap to other states (Eg. looping between ‘MAIN MENU” and “GAME OVER' ' constantly).

To conclude that the states were successfully rendered, and switches are possible, we used assertions (eg. AssertEquals and True). We followed up with verifying the preconditions of each variable according to the state, before allowing the state switch. We also created mock variables for player/enemy positions to ensure that the value being read is the same as the value of the game object. These were also done utilizing assertions. Last but not least, we determined that no background movement and unnecessary switching was taking place. These tests were separated according to the amount of game states we had (eg. approximately 7 tests in file).

ObjectSetter Test:

For the object setter class, we decided to test three key points: verifying that the correct object types are being generated, checking if the object being generated is generated at the correct location and making sure that the object's boolean variables are all set to the correct conditions.

For the first keyp point, we verified the object type using the object variable "name" and using assertEquals to check if the name matches for each object type. For example, for a reward object we set its name to "Coin". So, for checking if a reward object is being created, we simply used assertEquals with an expected value of "Coin" in the first parameter and passed the obj.name in the second (actual) parameter. This will make sure that when creating a new object, it is creating the correct type by checking if the names match. We used this method for every type of object in our game: reward, punishment, exit and bonus reward.

Secondly, for making sure all boolean variables are set correctly, we again used assertEquals by grabbing its boolean variables and comparing it with the expected values. We simply grabbed all of the object's boolean variables: obj.collision, hurtful and isExit, and passed each item as the second parameter and compared it with the expected boolean values: true or false. This method of checking boolean variables was used for all object types.

Lastly, for testing object positions, we also used assertEquals using the object's worldX and worldY variables which record its position on the map. We passed each variable as the second parameter and compared it with the expected values we set. We used this method for all the object types except for bonus reward. For bonus reward, since its position is not set, rather it can generate on any free tile, we decided to rather check the tile it was generated to for any other existing tiles. This included checking if the tile already contained an existing object: reward,punishment,exit, or if the tile is already set to be a wall tile. All these tests used assertNotEquals for verifying bonus reward positions.

Player Test:

For the player class, we decided to test the two most important functions for the player, which were movement and tile collision

For movement, we simply checked if the movement commands were being processed correctly by our code. We checked that if the player were to move "up", that its new position would be equal to expected position on the game screen. We used assertEquals to check if the updated position was correct and assertNotEquals to make sure that the player did not move in the wrong direction it was told (making sure it didn't move down, right or left instead of up).

Then for tile collision, we tested it by spawning a player on an existing wall tile on the game screen. Like with the movement, we used assertEquals to make sure that if the player sprite tried to move in any direction on the wall tile, it wouldn't move. We made sure then that since the player couldn't move, its position did not change.

Features that needed to be unit tested & brief description:

- Movement of player:
- We need to test that the player character moves in the direction that the user inputs (via keyboard up/down/left/right keys), i.e.) that the player's position changes by 1 adjacent cell unit in the respective x or y direction.
    - We need to test if there is a collision between the player and any of the walls/barriers so that the player does not move in that direction (not allowed as per game specifications).
- Collisions
    - Player with reward/punishments/enemies
    - Also test that player score updates properly
- Objects: We need to test that the when the objects are placed on the board at the start of the game, they have correct type specifications, booleans are all set to 'false' (as the game has not started and there haven't been any collisions yet), and that they are in the correct positions that were specified by our maze design.
        - Test for correct object types
        - Verify correct boolean settings
        - Verify positions if correct
- Change in game states
    - Test that player collision with exit triggers change in game state
- Interactions of game with keyboard & mouse
- Test game resets properly upon restart
    - Reset score
    - Reset timer
    - Reset player, objects, enemies and rewards

Identifying interactions between different system components:

- Interaction between the user interface and game states
    - Clicking 'main menu, 'new game,' 'resume game' buttons on the screen form part of the condition for updating the internal game state. ex) If the user clicks the 'new game' button from pause screen, this interaction will change the game state from 'paused' to 'playing.'

Features or code segments that are not covered in our tests:

- We did not create separate dedicated tests for GUI interactions (key & mouse events), since our code was written in such a way that we can test for the effects of key/mouse events without having to actually simulate the event in the test case.

- characterMovementTest (in PlayerTest.java): We tested that the player's position changes based on user inputs (via keyboard up/down/left/right keys) via this test method, rather than making a separate test for KeyHandler.
- GameStateTest.java: We tested that transitions between game states happened when all sufficient and necessary conditions were met. These conditions include, but are not limited to, mouse input. Therefore we did not need to test mouse events separately.

Results

GameStates Test:

- For the MENU state, we were able to conclude that the player, time, score, and enemy objects were not moving or acting in the background. Furthermore, we were able to validate that the mouseInputs were working when certain buttons/regions on the screen were clicked. In this case, the only options were to "Exit" switching to the Exit state successfully and the "Start Game" entering the "RESTART" phase into "PLAYING" state successfully.
- The next state is PLAYING, and there we were able to determine that the Player continues to play the game as long as enemyCollision is false. Furthermore, when the Player is has not collected 7 rewards it does not allow the exit tile to initiate "GAMEWIN" state. In addition, to test the score during the "PLAYING" state is not negative, we created a mock score in testing to ensure that the condition runs (eg. randomInteger must be greater than score).
- The following state tested was the "PAUSE" state, and similarly to MENU, we determined that objects did not continue to move in the background. Furthermore, we verified that the corresponding state did not switch due to unanticipated delayed/ background movement.
- The state tested next was the "GAMEOVER" state. For this we were able to conclude that player to enemy collision, and score hitting below zero successfully lead to this game state. In addition, we have to verify that the timer and score successfully stop iterating.
- For the "GAMEWIN" state, we successfully tested that the conditions of collecting 7 rewards, and touching the "EXIT" tile were met. In correlation, we found that the game players and enemy objects have stopped moving and are not touching each other.
- We did not test the "EXIT" state since that is self explanatory, but we also verified that the failure to meet each condition per state did not render the state switch.

ObjectSetter Test:

- For setting the object types, we were able to conclude that all correct object types were being correctly generated in the game. The correct objects were being generated on the

game screen with their correct images. We found zero problems where the game would accidentally deem for example, a reward object as a punishment object and vice versa.
- Then for checking each object's positions, we found that all the preset positions for rewards, punishments and exit all worked with zero errors. They were all being generated exactly where we wanted them to on the game screen. Additionally for the bonus reward, since it could spawn on any free tile randomly, we had to check if each tile it was generating on was a credible tile. In our testing we found zero errors with the method and deemed that the bonus reward could only spawn on free tiles.
- Lastly, for checking each object's boolean values it was evident that each boolean variable was being set to the correct value. The boolean values handle each object's collision and collision events so it was important that these variables were being set correctly. We concluded that each object had all their boolean values set correctly.

Player Test:

- Firstly, for testing player movement, we concluded that the player sprite was moving when movement commands such as up, down, left, up were passed into the player's movement function. The player sprite moved to the correct positions it was told with zero errors (i.e. when telling the sprite to move "up", it would move up).
- Next for testing player collision with the tiles, all tests were passed as expected.If the player tried to walk into a wall tile (a tile with collision), our test showed that the player would not be able to move into that tile from all directions. This showed that collision with the tiles was working.

Findings

We understood more about our code structure as we were writing tests. We also had to think about what the input space was for our methods and what boundary cases there were. Some of the findings we had include refactoring our code via variable integration. This was done by reducing redundancy in repeating lines, and dismissing code that was not actually being used in the game process. In addition, we added a few getters for testing which led us to reflect on the coupling and cohesion in our code. Adding the getters made us realize our code was loosely coupled in areas that we may not have noticed before. Through this process we also discovered simpler ways to organize the code with the usage of file/pathing and creating subfolders for specific classes. This made the process of understanding which class did what according to the folders they fell under (eg. states folder contained all state java files).