Explain such interactions in your report and create integration tests for them. Document which test case/class covers which feature/interaction in your report.

There are two main sets of interactions between different components of the game. The first one is the interactions between the objects of the game like how the collision between player, enemies, walls, and items affect each other. The other set is the interaction between different systems in the game like UI and sound effects and how these change and affect the game as the state of the game changes.

BackgroundTest.java Class

`loadMapTest()` - Checks if the map is loaded correctly with correct tiles.

BoardTest.java Class

Individual Unit Tests

`TimerTest()` - Checks if the timer correctly gets a time. Tested case: 1 second.

`isDeadTest()` - Puts an enemy in the same tile as the player, checking the collision between player and enemy and Checks the game over condition (player dying when collides with moving enemy).

`tickTest()` - Checks if the tick stops when the game is over.

`endScreenTest()` - Test to see if the end Screen code works correctly when the player has completed the game.

Integrated Tests with Player

`KeyUpTest()`, `KeyDownTest()`, `KeyRightTest()`, `KeyLeftTest()` - Tests for keyPressed events, moving the Characters in one of the four directions.

`scoreTest()` - Checks if the item pick up works correctly and the player collects points by picking up the items.

FalseKeyTest() - Checks if player does not Tested case: spacebar

punishmentTest() - Checks if score decreases once the player gets to the trap. Tested case: current score 2

punishmentTest2() - Checks if the player dies once the player gets to the trap with 0 score. Tested case: current score 0

Integrated Enemy Tests

enemyMovementTests - Checks if the enemy correctly moves to the player once it gets to the board.

CharacterTest.java

checkWallTest() – Checks the checkWall function, returning false if the tile is a wall and false if not. 3 Cases for the test: Tile is a wall, Tile is not a wall, Tile is out of bound.

upTest(), downTest(), leftTest(), rightTest() – Checks the collision between player and wall tiles for four directions. 2 Cases, if the neighbouring tile in the moving direction is a wall, or it is not a wall tile.

EnemyTest.java

EnemyTickTest() - Checks if the enemy's state is changed to “caught” once it collides with the player.

PlayerTest.java

KeyTest() - Checks if the player’s status changes to correct direction once gets an assigned keyboard input. Tested case: up key, down key, left key, right key

doorTest() - Checks the variable ‘isPlayerAtDoor’ which returns true when the player gets the opened door. Tested cases: player is at door and it is opened / player is not at door and it is opened / player is at door and it is closed / player is not at door and it is closed

tickTest() - Checks all the updates in player class.

a. Checks if variable ‘moves’ reset after 8 moves

b. `Checks if door opens after the player collects all the reward`
c. `Checks if score correctly`

Measure, report, and discuss line and branch coverage of your tests. Document and explain the results in your report. Discuss whether there are any features or code segments that are not covered and why.

Line Coverage:

1. BoardTest.java
   a. Line coverage = 179 / 210 x 100 = 85.23%, excluding import statements, getters and setters.
2. PlayerTest.java
   a. Line coverage = 126 / 151 x 100 = 83.44%, excluding import statements, getters and setters.
   b. Branch coverage = 100%
3. EnemyTest.java
   a. Line coverage = 194 / 205 x 100 = 94.63%
4. CharacterTest.java
   a. Line coverage = 100%
5. BackgroundTest.java
   a. Line coverage = 64 / 134 x 100 = 47.76%, excluding import statements.

Branch Coverage:

For some aspects of our code, we have had only 50% branch coverage. In Board.java, we could've further tested the bonusTimer and isDead variable to make sure that we have a guaranteed 100% branch coverage. The integrated KeyTests for Player have a 100% branch coverage, since we tested corrected inputs and incorrect inputs for keyDown events and checked whether the player was moving as desired.

In Player.java, the branch coverage for the keyTests were 50%, since we only tested the correct inputs for KeyPressed events. In tickTest, there is a 100% branch coverage, since the tested values cover the false and true values for the statements. For doorTest, there is 100% branch coverage again.

In Character.java, the branch coverage for checkWall is 100%.

Things not covered

a. We couldn't test the functions and lines that are related to the "image" and "sound" as it is hard to check if the images are correctly loaded using JUnit. Rather, we checked variables that showed the status of the class such as "sprite".
b. We also couldn't check the "random" functions. There is no way to check if it is real random points. It might be able to check the bound, but for that test we should do a repeated test. I consider that as an inefficient test.
c. We didn't test whether every key pressed on a keyboard will have a reaction for the player, as it seemed inefficient and redundant.

Briefly discuss what you have learned from writing and running your tests. Did you make any changes to the production code during the testing phase? Were you able to reveal and fix any bugs and/or improve the quality of your code in general? Discuss your more important findings in the report.

While writing and running the tests, we needed to carefully check the codes that are written by other people. During this, we did not only find some small logical errors, but also realized that some codes are tightly coupled and lowly cohesive.

The first thing we did was refactor our codes. We divided our codes into three categories: control, model, and view. We also made a Main class, separated from Game class. Secondly we added getters and setters so that tests and other classes do not have to directly access the variables. We also changed some variable names. Those three changes were for the improvement of code quality.

After reform the code, we changed the score system which had a slight logical error. It is found during the test. Besides that, we make our end screen better for the quality of the game.

While testing the enemy movement, we realized that there are some missing aspects for it to be more successful for tracking down the player. While the enemy moves closer to the player at every turn, it can sometimes get caught up on walls and have trouble navigating around them. Testing the movement revealed that the enemy is sometimes not recognizing that there is a wall next to it, and tries to walk through the wall. After testing we improved the movement with the walls in consideration.

While testing the Player class, we realized that it is too much of a “God” class, as in it does too much work and it could be divided into smaller modules.

Overall, we learned that writing the code in a way that is easily testable is a much more efficient and beneficial way of writing software. We also learned that having classes that are too dependent on other separate classes and having one class that has one main method to do everything is bad code design.