One of the main functional features in games is collision detection. Players can earn points when encountering objects, deduct points at fire objects and terminate the game when encountering enemies. This unit test mainly focuses on CollisionChecker's collision detection. The main prerequisite for testing is the launch of the game panel gamePanel, as well as the initialization of players, obj, and enemy. At the same time, the core logic of collision detection is the position of players, obj, and enemy. Based on this, we have designed the following testing plan:

1. Test if the default collision settings at the beginning of the game are correct: player, objects, and enemy all have collision switches, and these switches need to be initialized correctly to ensure their effectiveness during the detection process.
2. Test the collision between players and coins, cash, and multiple objects, and check if the score is correct: mainly by modifying the position of players and obj to achieve collision, and then test whether the collision results and switches are correctly modified after collision detection. For objects like obj, players need to be assigned scores.
3. Test the collision between players and enemies, and check if the game is over: mainly by modifying the positions of players and enemies to achieve collision, and then test whether the collision results and switches have been correctly modified after collision detection. For objects like enemy, it is necessary to set the isEnd property of gamePanel to terminate the game.
4. Test the collision between players, coins, and enemies, but this time there is some displacement of the player's position to check the flexibility of collision detection. The previous test was to make the player, obj, and enemy in the same position to achieve collision. This test further achieved edge position touch detection, making the game more realistic and reliable.

For Unit and Integration Test, we decided to test a multitude of features such as object instantiation, variable change, and key input. AssertSetterTest ensures that the reward objects and enemy objects are properly set in the indicated locations. CellManageTest checks that proper variables are set for the map and there is no error in instantiation by

using assertNotNull(). KeyCatcherTest test for variable setting given a key input. PlayerTest tests for proper instantiation of the player object. StartPanelTest and UITest tests for the proper display of messages for users and ensure the graphical interface is behaving as expected. The text, color and size of the interface are all tested using assert to ensure proper behavior.

GamePanelTest tests the update function that takes in a key input and the collisionOn boolean value. It covers all the combinations of conditions such as keyPress being up and collisionOn being true, keyPress being up and collisionOn being false, keyPress being down and collisionOn being true and so on for all combination between keypress (Up, Down, Left, Right) and collisionOn (true,false). This ensures that the player keyboard input is properly processed given the crucial variable of collisionOn being true or false. This acts as an Integration Test to ensure proper interaction between the methods that set variables and the user key input.

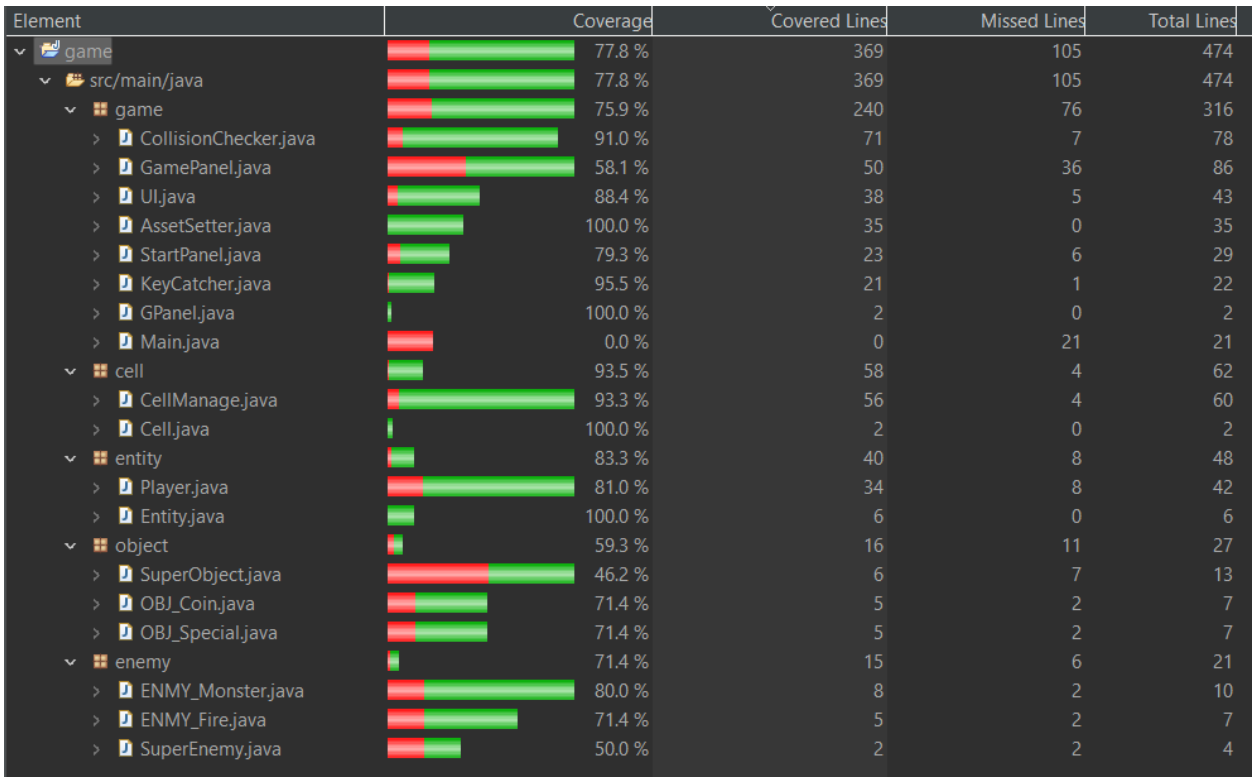
To ensure the quality of the test cases, we create assertions for all the objects created and their corresponding values as the overall quantity of objects is not large. For example, in AssertSetterTest we create assertions for the x and y variables of all 10 objects created to ensure all objects are properly instantiated.

The above test plan has been executed, and the final test result is: passed all tests. The collision detection function can be correctly implemented and meets the expected results. Harvest and Experience: Through this test, we can see that unit testing of key features in the Harvest and Experience: Through this test, we can see that unit testing of key features in the game is very necessary. This can ensure that the basic functions of the game run normally, and game is very necessary. This can ensure that the basic functions of the game run normally, and also quickly check whether the original collision detection logic also quickly check whether the original collision detection logic is affected when modifying or is affected when modifying or adding new features. Meanwhile, although collision detection is only a positional judgment, its adding new features. Meanwhile, although collision detection is only a positional

judgment, its results will have an additional impact on the progress of the game. During the testing process, its results will have an additional impact on the progress of the game. During the testing process, it is necessary to strictly judge is necessary to strictly judge the side effects to ensure that the achieved effect meets the side effects to ensure that the achieved effect meets expectations.

Most of our tests are written for existing codes and it is difficult to write failing tests when all the codes are already implemented and easily accessible. We continuously improve our game based on our initial plan such as object collision but our test cases did not directly reveal any bugs and were mostly written to confirm proper behavior of our improvements. Our initial coverage test was around 72% but this helped us identify some unused codes and we were able to clean up the duplicate codes while maintaining proper game function.

The final coverage data of the tests with 77.8% coverage are shown below:



Element	Coverage	Covered Lines	Missed Lines	Total Lines
game	77.8 %	369	105	474
src/main/java	77.8 %	369	105	474
game	75.9 %	240	76	316
CollisionChecker.java	91.0 %	71	7	78
GamePanel.java	58.1 %	50	36	86
UI.java	88.4 %	38	5	43
AssetSetter.java	100.0 %	35	0	35
StartPanel.java	79.3 %	23	6	29
KeyCatcher.java	95.5 %	21	1	22
GPanel.java	100.0 %	2	0	2
Main.java	0.0 %	0	21	21
cell	93.5 %	58	4	62
CellManage.java	93.3 %	56	4	60
Cell.java	100.0 %	2	0	2
entity	83.3 %	40	8	48
Player.java	81.0 %	34	8	42
Entity.java	100.0 %	6	0	6
object	59.3 %	16	11	27
SuperObject.java	46.2 %	6	7	13
OBJ_Coin.java	71.4 %	5	2	7
OBJ_Special.java	71.4 %	5	2	7
enemy	71.4 %	15	6	21
ENMY_Monster.java	80.0 %	8	2	10
ENMY_Fire.java	71.4 %	5	2	7
SuperEnemy.java	50.0 %	2	2	4

The main concerns for the coverage is the GamePanel.java with 36 missed lines, Main.java with 21 missed lines and SuperObject.java with 7 missed lines (not large amount but overall coverage is only 46.2% for the file)

The GamePanel.java with 36 missed lines is mainly due to the graphical aspect of the game like drawing the enemy components onto the map as well as the exception catching lines which should not be executed when the code is working. The graphical aspect can be easily verified when the whole game is run (system test) and there are smaller unit tests in place like the enemy assertions in AssertSetterTest and assertions for the map in CellManageTest to test the proper instantiation of those individual elements.

The entire Main.java with 21 missed lines is not covered by the test cases as it is a collection of object creation and integration and can also be easily verified when the whole game is run (system test). Most of the lines involve making an instance of some other class which are tested in our test suite and our overall test for the game shows proper interaction of the tested components.

The SuperObject.java with 7 missed lines is not a lot of lines but the file is small as well and it is concerning that the coverage is only 46.2%. The missed lines include exception catching and drawing images. As mentioned above, exceptions are not meant to happen and the images are easily verified by running the game.