GROUP 11
Nicholas Lagasse
Ekansh Chawla
Felmer Macadangdang

2.1 - Unit and Integration Tests

For our unit tests, we decided to only write unit tests for methods that had parameters, because methods with no parameters were either calls to other classes which would be tested elsewhere, or were methods to be used in methods with parameters.

Although constructors would naturally fall under this category, we decided to only refactor them by inserting try catch statements as well as null checking, and throwing relevant errors, as all of our constructors only dealt with assigning local variables.

We grouped the unit tests into equal thirds, with our names above each grouping of files we wrote tests for respectively.

UNIT TESTS:

Nicholas Lagasse
Main.java: none
BaseThread.java: none
GamePanel.java: isButtonClicked()

- Test button clicked method. Did not test paintComponent(), drawStartScreen, drawButton(), because they simply draw a passed graphics object at the coordinate.

GameState.java: saveOriginalCellContent(), setDifficulty(), updateSamuraiDamage(), spawnAdditionalSamurai(), isOccupied(), isEnemyAt(), isAdjacentToPlayer(), movePlayer()

- These methods need to be tested because they deal with setting present game state values, as well as present game state checking, which could have incorrect cases.

KeyHandler.java: none
Renderer.java: render()

- This method needs to be tested because passed gameObjects could be incorrect or be apart of an invalid game state

Window.java: none

Ekansh Chawla
GameObject.java:

- isSolid(): check if correct solid state is returned for different GameObject types.
- getTypeId(): ensure that each object correctly returns its assigned typeID
- getX(), getY(), setX(), setY() - validate that object position updates/sets after calling setter/getter methods.

GameObectFactory.java: createObject(String type, int x, int y)- confirm that object are correctly created based on type and initialized with the given coordinates.

Barrier.java:

- Verified that barrier is initialized as solid and retains coordinates and typeID
- Tested interaction logic.

Integration Tests:

- Maze Builder Validation - Verified that the maze initializes correctly with all cells populated and contain gameobjects.
- Enemy Interaction - Ensured that enemies move closer to the player as expected
- Player Score Updates - confirmed that the players score increases upon collecting a Bonus Item
- GameState Initialization - Checked that game initializes properly with game objects and counters also starting properly.
- Enemy Placement Validation - validated that enemies do not overlap during initialization
- PlayerHealth Behaviour - ensured that player health doesn't drop below zero after taking damage
- End Tile Placement - verified that the maze included an end tile for game completion.

<u>Felmer Macadangdang</u>

BonusItem.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position.

End.java: Constructor()

- The endpoint in the maze was tested to ensure proper position.

Ground.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position. Tested solid() property to ensure ai works properly.

Hole.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position.

MandatoryItem.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position. Mostly tested through integration tests.

Player.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position.

Wall.java: Constructor()

- Tested to ensure correct id was set and to ensure proper position. Tested solid() property to ensure ai works properly.

Samurai.java: Constructor(), moveTowardsPlayerAvoidingWalls(player1, testMaze)

- This method is the main method of Samurai ensuring that the Samurai moves towards the player and is able to get to the player. Tests were passed flawlessly.

MazeBuilder.java: none

- Tested through integration tests.

2.1.1 - Test Automation
See README.md

2.1.2 - Test Quality and Coverage
In our testing, we first recognized any and all constructors, and put try catch statements in them. In phase 2, we did not have these try catch blocks to catch any null passed objects, but after adding them we decided not to write test cases for them, since the constructors called other methods which we would later test in smaller test files.

For writing our test cases, we tried to pass as many edge cases as possible to the tested methods, as that is where errors typically occur. Some tests had multiple different combinations of edge cases, so we tried to write as many combinations of them as possible.

The below figure (Figure 1.) shows all of the line and branch coverages given from IntelliJ.

Some classes show 0% on all coverages (BaseThread, GamePanel, KeyHandler, Main,Window), as these methods were starting points for our code, and simply called multiple beginning functionalities of all of the other classes. We did not need to write tests for these classes.

On average our code shows that line coverage was 49%, and branch coverage was 47%. This is because we did not write tests for every single piece of functionality in our code, such as all of the previously mentioned 0% coverage files, or basic methods such as getters, setters, and equality checkers. We did not deem these methods to be in need of unit testing.

Without these reductions to the percentages found, we still would not have had 100% line and branch coverage, because we did not cover every single possible test case, due to prioritizing other aspects of phase 3 such as refactoring.

Element	Class, %	Method, %	Line, %	Branch, %
project11	73% (17/23)	60% (82/135)	49% (270/551)	47% (121/255)
Barrier	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
BaseThread	0% (0/1)	0% (0/5)	0% (0/32)	0% (0/8)
BonusItem	100% (1/1)	50% (1/2)	50% (1/2)	100% (0/0)
Constants	100% (1/1)	77% (14/18)	84% (21/25)	100% (0/0)
End	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
Enemy	100% (1/1)	85% (6/7)	72% (13/18)	33% (2/6)
GameObject	100% (1/1)	100% (7/7)	100% (11/11)	100% (0/0)
GameObjectFactory	100% (1/1)	100% (1/1)	80% (8/10)	66% (6/9)
GamePanel	0% (0/2)	0% (0/9)	0% (0/102)	0% (0/30)
GameState	100% (1/1)	75% (22/29)	66% (100/151)	55% (53/95)
Ground	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
Hole	100% (1/1)	50% (1/2)	50% (1/2)	100% (0/0)
Item	100% (1/1)	50% (3/6)	62% (5/8)	100% (0/0)
KeyHandler	0% (0/1)	0% (0/5)	0% (0/17)	0% (0/16)
Main	0% (0/1)	0% (0/1)	0% (0/8)	100% (0/0)
MandatoryItem	100% (1/1)	50% (1/2)	50% (1/2)	100% (0/0)
MazeBuilder	100% (1/1)	100% (9/9)	98% (64/65)	89% (43/48)
Player	100% (1/1)	50% (2/4)	42% (3/7)	0% (0/2)
Renderer	100% (1/1)	23% (3/13)	37% (16/43)	58% (10/17)
Samurai	100% (1/1)	71% (5/7)	59% (19/32)	29% (7/24)
Wall	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
Window	0% (0/1)	0% (0/1)	0% (0/9)	100% (0/0)

Figure 1. *Test Coverage*

2.1.3 - Findings

We learned a lot about our initial code from phase 2, and how much we really had to refactor. We noticed at first that a lot of code needed null/bound checking before working with the passed parameters. We also found that our code had multiple cases of redundant classes, where the methods defined in them were not used. We deleted these unused classes, and made sure the intended methods belonged to the intended classes.

Alongside assignment 4, we managed to refactor a good amount of our code, grouping together common functionality, as well as putting all game variables and constants into a single file with getters and setters for easy modularity and readability.

As well as refactoring and increasing the quality of our code, we were also able to find memory errors in the code which we did not find in phase 2, namely an enemy and player trying to move into the same game tile. We managed to identify and correct these sections of code to fix the errors due to the amount of test writing we did.