

## Phase 3 Report

### 2.1 Unit and Integration Tests

**Briefly explain the features that need to be tested. Document which test case/class covers which feature/interaction. Explain interactions between different components of the system.**

Features Tested:

- Player is set in starting position (Feature was added in Player constructor)
- Player moves in every direction with a valid key press (Feature was added in Player's update and KeyHandler class)
- Player doesn't move with invalid key press (In Player's update and KeyHandler class)
- Player colliding with wall in every direction (In CollisionChecker's checkTile)
- Player score changes when they collect a reward or punishment (In Player's update and pickUpObject)
- Player loses when they make contact with an enemy (In Entity's (Superclass) update and CollisionChecker's checkEntity and checkPlayer)
- Player loses when their score becomes negative (In Player's update and pickUpObject)
- Enemy is placed (In AssetSetter's placeEnemy)
- Enemy moves toward player (In Node and Pathfinder classes for A\* Search, Entity's searchPath)
- Punishment is placed (In AssetSetter's setObject)
- Rewards are placed (In AssetSetter's setObject)
- Bonus is placed (In AssetSetter's setObject)
- Bonus spawns and despawns (In AssetSetter's setSpawnTime)
- Rewards and Bonus cannot be picked up by enemies (In CollisionChecker's checkObject)
- UI can receive messages and store them (In UI's addMessage)

Player needs to be at the starting tile, as per specification requirements. Movement must only occur with valid key presses in order to allow a user to play the game properly. Various game components needed testing for their collision detection, interaction based on collision, changes based on game state, and their initial placement. Enemies also need to be properly placed on the board and have their pathfinding algorithm tested. Bonus reward must have its spawn time and location randomised. Bonuses must also despawn after their despawn time (spawn time +

20) is exceeded. UI must be able to receive messages and display them with respect to system variables such as time and score.

Testing classes:

- AssetSetterTest - Unit test to validate Asset Setter inputs, while ensuring objects are placed correctly and not null
- CollisionTest - Test the player's collision with walls in every direction
- EnemyTest - Tests the creation of enemies (enemyNotNULL), placement of punishments (trapsPlaced), pathfinding (movingToPlayer), and ignoring reward/punishment collision (enemyNoPickupTest)
- GamePanelTest (Currently doesn't do anything)
- MapTest - Tests the placement of background tiles (testTilePlacement), and the placement of individual walls (countWalls)
- MovementTest - Tests the player's movement in every direction
- PlayerTest - Tests miscellaneous player interactions, such as achieving a negative score (playerNegativeScoreTest), getting hit by an enemy (IntersectEnemyTest), and getting the first reward (GetFirstRewardTest)
- RewardTest - Tests various interactions with player and the rewards, such as picking up all rewards and winning the game (allBaseRewards), picking up all rewards and the bonus and winning the game (allRewards), and verifying the bonus reward spawn window
- UITest - Tests the addMessage method (addMessageTest), so that the UI can display messages to the user

Everything is connected through the GamePanel class, as every visible interaction in the system happens within this panel. Player's pickUpObject method interacts with the UI to add messages, sound to play sound effects, and the GamePanel itself to change game states if need be. Player also interacts with the CollisionChecker to check its surrounding tiles, objects, and enemies. Enemies and objects also have collision detection from the CollisionChecker class so that they may interact with the player. Every object and enemy also interacts with the AssetSetter class in order to be placed around on the board and initialised. When the player picks up their first reward, The UI class changes what it draws depending on the GamePanel's game state.

### **2.1.1 Test Automation**

**Complete the README File with complete instruction on how to build, run, and test the game.**

READ ME CONTENTS:

*\*\*How to build the project:\*\**

**1. Clone the repository**

\*\*\*

```
git clone
```

```
git@csil-git1.cs.surrey.sfu.ca:cmpt276s23_group1/group-project.git
```

```
cd group-project/PhaseTwo
```

\*\*\*

**2. Build the project with Maven (also executes tests):**

\*\*\*

```
mvn package
```

\*\*\*

*\*\*How to run the project:\*\**

- Run the .jar with java:

\*\*\*

```
java -jar target\PhaseTwo-1.0-SNAPSHOT.jar HeistGame.java
```

\*\*\*

*\*\*How to test the project independently of build:\*\**

- Test the project with Maven:

\*\*\*

```
mvn test
```














































\*\*\*

### **2.1.2 Test Quality and Coverage**

**Discuss measures taken for ensuring the quality of test cases. Calculate coverage for lines and branches. Explain results in the report. Discuss whether there are any features or segments that are not covered and why.**

To ensure the quality of our test cases, we first discussed what features truly needed testing, as well as what features could likely be put in the same class. Afterwards, we discussed the different branches that could result from our conditional statements. There is a lot of code in our project, but many of the methods are private and do not need to be tested. We only need to test the public methods as they create interactivity in the game and call the private methods. If we tested every method, the process would be incredibly tedious, and we would create redundant tests.

**Calculate coverage for lines and branches**  
**Results**

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
PhaseTwo	 54.6 %	6,707	5,578	12,285
src/main/java	 43.1 %	4,009	5,303	9,312
groupOne.game	 43.1 %	4,009	5,303	9,312
UI.java	 6.8 %	145	1,976	2,121
AssetSetter.java	 18.4 %	333	1,472	1,805
KeyHandler.java	 22.2 %	119	417	536
GamePanel.java	 38.4 %	249	399	648
Entity.java	 38.2 %	218	353	571
TileManager.java	 62.8 %	247	146	393
CollisionChecker.java	 87.8 %	837	116	953
Enemy_Dog.java	 44.2 %	80	101	181
Player.java	 84.0 %	521	99	620
Enemy_Guard.java	 69.1 %	125	56	181
HeistGame.java	 0.0 %	0	40	40
Sound.java	 77.3 %	99	29	128
Pathfinder.java	 95.3 %	484	24	508
UI_Information.java	 60.0 %	36	24	60
OBJ_Door.java	 0.0 %	0	19	19
OBJ_Laser.java	 0.0 %	0	13	13
UI_Button.java	 96.1 %	74	3	77
UI_MenuArrow.java	 88.9 %	24	3	27
UI_PausedSign.java	 88.9 %	24	3	27
UI_Reward.java	 94.3 %	50	3	53
UI_TitleBackground.java	 90.9 %	30	3	33
OBJ_Bonus.java	 93.8 %	15	1	16
OBJ_Damage.java	 93.8 %	15	1	16
OBJ_Gate.java	 93.8 %	15	1	16
OBJ_Reward.java	 97.1 %	34	1	35
Helper.java	 100.0 %	27	0	27
Node.java	 100.0 %	9	0	9
S_Entity.java	 100.0 %	83	0	83
SoundNames.java	 100.0 %	81	0	81
Tile.java	 100.0 %	6	0	6
UI_Object.java	 100.0 %	29	0	29
src/test/java	 90.8 %	2,698	275	2,973
groupOne.game	 90.8 %	2,698	275	2,973
CollisionTest.java	 89.0 %	519	64	583
PlayerTest.java	 90.3 %	522	56	578
MovementTest.java	 84.3 %	257	48	305
EnemyTest.java	 89.1 %	301	37	338
RewardTest.java	 97.4 %	736	20	756
AssetSetterTest.java	 84.2 %	101	19	120
UITest.java	 83.3 %	60	12	72
GamePanelTest.java	 89.6 %	95	11	106
MapTest.java	 93.0 %	107	8	115

Coverage was calculated through the Eclipse IDE, though the values are not entirely accurate due to a lack of constraints. For example, our assertions did not have 100% condition coverage because our system's behaviour for false values is undefined. Through the coverage report, we also spotted several more instances of dead code that may be removed in the future. For example, in `Enemy_Dog` and `Enemy_Guard` classes, there was an else statement that would've made enemies move in random directions when blinded, but since we did not end up adding that functionality the code remains dead.

### **Any segments not covered and why**

Key handling from PAUSE, WIN, and LOSE states were not tested as we were primarily focused on key inputs for PLAY state. However, all 3 of the possible test cases are valid and may be added in the future. Private methods were not tested

directly, though unit tests were performed on the relevant components of each method. We did not write any unit tests for the A\* Search pathfinding algorithm, though it was included in several integration tests and meets requirements. For testing purposes, only map 1 was used as it has the simplest layout. However, by equivalence partitioning, we can assume that the tests would pass for all map inputs. Components included in our main driver, such as Graphics2D and JFrame were not covered because the main method is not executable through tests.

### **2.1.3 Findings**

**Discuss learnings from writing and running tests. Did we make any changes? Were we able to reveal and fix bugs? Were we able to improve code quality and clarity?**

From writing and running tests, we discovered many small bugs and made many changes in order to fix all of them. For example, the spawn location method for the bonus reward was designed to be random, but would only attempt to find a location while the player was in the title screen. There were very rare cases where the bonus reward would spawn in a wall because the system could not find an open tile before the user pressed the play button. If we only tested from the user's perspective, and did not automate our tests, we would not have been able to find the bug. Through proper testing and refactoring, we easily traced the faulty method and updated it to find a valid tile during the entire time the bonus reward is not able to be picked up. Additionally, we added initial values so that the bonus reward can spawn in an open tile, in the extremely unlikely event that the spawn location method cannot find a valid tile in time. Many other fixes were made, similar to the example stated, that fine-tuned our project and made the system function better and the code much more clear. By analysing the coverage of our tests we got a better understanding of the underlying logic flow of our system and spotted new areas for future improvement.