

## The MazeGame Phase 3 Report

### Unit and Integration Tests

Feature / Interaction	Explanation	Covered by
Basic Movement for MovingCharacter (Player or Enemy)	A MovingCharacter attempts to move to a) a solid cell b) an empty cell	PlayerTest.java moveToSolidCell() moveToEmptyCell()
Win Conditions	The Player reaches the end cell having a) collected all regular rewards b) missed some regular rewards	PlayerTest.java reachEndAllRegularRewardsCollected() reachEndSomeRegularRewardsRemaining()
Reward Encounter (Player)	Player encounters a reward	PlayerTest.java touchReward()
Post Reward / Punishment Encounter (Player)	Player leaves the cell that contained a reward / punishment	PlayerTest.java leaveReward()
Punishment Encounter (Player)	Player encounters a punishment and the resulting score is a) non-negative b) negative	PlayerTest.java touchPunishmentNonNegativeScore() touchPunishmentNegativeScore()
Enemy Encounter (Player)	Player encounters an enemy	PlayerTest.java touchEnemy()
Bonus Reward creation	Bonus reward attempts to spawn when: a) there is no existing bonus reward (should spawn a new one) b) a bonus reward already exists (should not spawn a new one)	BonusRewardTest.java createBR() creatBRWithExistingBR()
Bonus Reward Expiration	Bonus Reward expires (should remove the bonus reward)	BonusRewardTest.java checkExpire()



































Bonus Reward Encounter	Bonus Reward is touched by the player (should be removed)	BonusRewardTest.java CollectBRTTest()
Enemy movement	Enemy move towards Player current location, choosing the closest path up, down, left, or right	EnemyTest.java getCurrentSquared DistanceToPlayerTest() MoveTest()
Enemy encounter Regular reward/ punishment/ bonus reward	Enemy moving into a cell that contains Regular reward/ punishment/ bonus reward	EnemyTest.java EnemyTouchRegular RewardTest() EnemyTouchPunishment Test() EnemyTouchBonus RewardTest()
Enemy leave Regular reward/ punishment/ bonus reward	Enemy moving away from a cell that contains Regular reward/ punishment/ bonus reward	EnemyTest.java EnemyLeaveRegular RewardTest() EnemyLeavePunishment Test() EnemyLeaveBonus RewardTest()

## Test Quality and Coverage

Our team used verification procedures to ensure the built system is behaving correctly according to the specifications. We also ensured the test code quality by using JUnit as the testing framework for our project's automated tests. The tests were built based on structural-based testing, where the set of techniques that use the structure of the source code were used as the primary artifact for guidance. Testing was approached at the unit level, where we created unit tests with easily controllable parameters. We also assessed our test suite's quality by using the JaCoCo Maven plugin to calculate test code coverage.

We used public methods (move) to test private methods (getBestMove for the enemy) to reduce test case duplication, tested getter and setters through other methods and used equivalence partitioning to reduce the number of test cases. For each test class, we first identified the parameters for to-be-tested classes and methods. Then, we derived each parameter's characteristics and added constraints in the tests to minimize the test suite. Combinations of the input values are then generated with considerations of variations in conditions.

**Figure 1. JaCoCo Test Coverage Results**

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Initialize		0%		0%	5	5	43	43	1	1	1	1
Enemy		86%		64%	15	35	22	115	1	10	0	1
Player		82%		62%	18	40	25	98	3	11	0	1
EndScreen		0%		n/a	2	2	13	13	2	2	1	1
BoardState		90%		81%	14	62	16	131	0	17	0	1
MovingCharacter		73%		43%	9	14	12	47	2	6	0	1
LostScreen		0%		n/a	2	2	9	9	2	2	1	1
BonusReward		76%		50%	1	5	4	22	0	4	0	1
Initialize.new WindowAdapter().{...}		0%		n/a	2	2	3	3	2	2	1	1
Main		0%		n/a	2	2	3	3	2	2	1	1
Board		92%		100%	2	11	3	27	2	8	0	1
TimeCounter		98%		100%	1	6	1	41	1	5	0	1
ScoreBoard		100%		100%	0	5	0	42	0	4	0	1
Cell		100%		83%	1	17	0	28	0	14	0	1
Punishment		100%		50%	1	4	0	14	0	3	0	1
Rewards		100%		100%	0	4	0	14	0	3	0	1
Position		100%		n/a	0	5	0	10	0	5	0	1
Total	542 of 2,972	81%	77 of 244	68%	75	221	153	659	18	99	5	17

We achieved an overall branch coverage of 68% with most of our classes having around 80% branch coverage.

Feature / Code Segment Not Covered	Reason
Initialize.java (game loop) Player keypresses	These features and code segments require user input and were manually tested by playing the game.
IOException catch blocks for reading images	These code segments only execute if our image files cannot be found and are not related to the actual game logic.
GUI (winning screen, losing screen, icons for game objects)	These features were tested manually by playing the game and indirectly by testing the game states (automatic).

## Findings

From writing and running our tests, we have realized that the game still requires substantial refinements. We found that singletons made unit testing more complicated since we could not create a fresh new instance for every test in a single run. GUI testing was also challenging to implement, as testing without having a screen to check visually and relying on checking values feels impossible at this stage.

The following list includes some refactor considerations that could improve code quality:

- Enemy object can only be spawned on free cell (not solid and does not contain any other object)
- Merge `calculateSquaredDistance` and `calculateSquaredDistanceWithEnemyCheck` into one function to reduce duplicate code.
- Create a new function `getRandomFreeCell` for `BoardState` constructor to reduce duplicate code and easier creation of other objects in the maze.
- Making `getBonusReward()` an array list so testing the function will be easier.
- Extending the GUI screens, create a `StartingScreen` that enables users to start the game and review game instructions, or a `PauseScreen` that displays game status to increase desirability and user interaction.

We managed to discover and fix a bug regarding enemy spawning on solid cells during the test phase. As we do code review, we recognized some code components that could be refactored in order to improve the code quality. However, following the rule of three and considering the time constraint for our project, we decided to avoid refactoring the other components as we approach the deadline.