



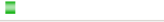
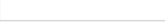










Issue with testing:

1. In adhering to the principles of the testing pyramid, our predominant testing approach centered around manual testing, primarily due to the nature of our Java-based game built with Java Swing, which involves graphical interfaces and windows.



























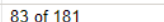

## Board

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● <a href="#">recursiveBacktracking(int, int, Stack)</a>		88%		86%	9	61	13	101	0	1
● <a href="#">updateTerrain()</a>		100%		91%	1	7	0	14	0	1
● <a href="#">Board(int, int, int, int)</a>		100%		n/a	0	1	0	10	0	1
● <a href="#">printMap()</a>		100%		100%	0	3	0	6	0	1
● <a href="#">validSpawn(int, int)</a>		100%		100%	0	3	0	4	0	1
● <a href="#">spawnEntity(int, int, int)</a>		100%		100%	0	2	0	4	0	1
● <a href="#">getMap()</a>		100%		n/a	0	1	0	1	0	1
Total	65 of 783	91%	16 of 136	88%	10	78	13	140	0	7

The board class is responsible for generating the 2D matrix for the map by utilizing the recursive backtracking algorithm .

1. Generating mazes of varying sizes
2. Checking for boundary conditions such as spawning a maze of size 0
3. Checking if map entities are all being spawned




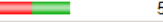






## BaseReward

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● <a href="#">draw(Graphics2D, GamePanel)</a>		0%		0%	2	2	6	6	1	1
● <a href="#">loadFrames(String, int)</a>		40%		50%	1	2	2	8	0	1
● <a href="#">setRewardValue(int)</a>		0%		n/a	1	1	2	2	1	1
● <a href="#">setRequired(Boolean)</a>		0%		n/a	1	1	2	2	1	1
● <a href="#">increaseScore(Game, int)</a>		0%		n/a	1	1	2	2	1	1
● <a href="#">changeFrame()</a>		89%		75%	1	3	1	7	0	1
● <a href="#">getRewardValue()</a>		0%		n/a	1	1	1	1	1	1
● <a href="#">BaseReward(int, int, int, int, int, int[])</a>		100%		100%	0	2	0	8	0	1
● <a href="#">getCurrentFrame()</a>		100%		n/a	0	1	0	1	0	1
● <a href="#">setCurrentFrameIndex(int)</a>		100%		n/a	0	1	0	2	0	1
● <a href="#">setBufferedImage(BufferedImage[])</a>		100%		n/a	0	1	0	2	0	1
● <a href="#">updateAnimation()</a>		100%		n/a	0	1	0	2	0	1
● <a href="#">getIsRequired()</a>		100%		n/a	0	1	0	1	0	1
● <a href="#">getCurrentFrameIndex()</a>		100%		n/a	0	1	0	1	0	1
Total	83 of 181	54%	4 of 10	60%	8	19	16	45	5	14

To test base rewards we tested:




1. Testing the constructor
2. Checking if the constructor contains an invalid reward value
3. Checking if the animations are going through the frames
- 4.

## BonusRewards

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
draw(Graphics2D, GamePanel)		0%		0%	2	2	6	6	1	1
loadFrames(String, int)		40%		50%	1	2	2	8	0	1
changeFrame()		89%		75%	1	3	1	7	0	1
BonusRewards(int, int, int, int, int, int)		100%	n/a	n/a	0	1	0	5	0	1
calculateBonus(int)		100%	n/a	n/a	0	1	0	3	0	1
getCurrentFrame()		100%	n/a	n/a	0	1	0	1	0	1
updateAnimation()		100%	n/a	n/a	0	1	0	2	0	1
Total	68 of 156	56%	4 of 8	50%	4	11	9	32	1	7

Bonus rewards follows the same test cases as BaseRewards














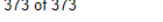
## CollisionChecker

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
checkTile(MainCharacter)		0%		0%	13	13	30	30	1	1
CollisionChecker(GamePanel)		0%	n/a	n/a	1	1	3	3	1	1
Total	302 of 302	0%	21 of 21	0%	14	14	33	33	2	2

CollisionTest.java were used to test the box collision methods in character.java (all the collision and movement logic was done here). These tests were especially useful because it helped use find the errors with the collision logic! For these tests we tested on point and off point.

1. testInBounds()
  - a. This was used to test for if a point is between 2 other points
2. testInBox()
  - a. This tests for collisions on a 1d line, it tests for cases where boxes overlap, and when a box is within another box
3. testBoxCollision()
  - a. This makes sure the boxes don't touch at all on the x and y planes, tests many different inputs as well as a box being within another box

## MainCharacter

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
MainCharacter(int, int, int, int, KeyHandler, GamePanel)		0%	n/a	n/a	1	1	25	25	1	1
move()		0%		0%	11	11	19	19	1	1
loadFrames(String, int)		0%		0%	2	2	8	8	1	1
getCurrentFrame()		0%		0%	5	5	10	10	1	1
action()		0%		0%	4	4	11	11	1	1
draw(Graphics2D)		0%		0%	2	2	4	4	1	1
changeFrame()		0%		0%	2	2	5	5	1	1
getMcKey()		0%	n/a	n/a	1	1	1	1	1	1
Total	373 of 373	0%	40 of 40	0%	28	28	83	83	8	8

In our testing suite for the MainCharacter class, we just tested its functionality and behavior. The tests were simple as most of the movement logic was testing with CollisionTest instead.

1. TestInitialization focused on validating the initialization of a MainCharacter object by checking key attributes such as position, dimensions, and speed.
2. TestMovement test simulated key presses for various directions, ensuring that the character's position was appropriately adjusted in response.

3. TestAnimation test verified the correct transition of animation frames during character movement.
4. Although a testCamera method was present in the test suite, it was commented out.

## 2.1.2 Test Quality and Coverage

In creating our tests, we strove to create quality test cases. We did so by making sure there were no redundant tests, for example by doing boundary tests, and trying to cover as many lines, branches, and conditions as we could. As shown above, we had issues achieving wide coverage of our code whilst testing our game. This is because of a few factors:

### 1. Abundance of private methods

As we learned from our lectures, you don't usually test private methods. This is because you either cannot access them from your test classes or you have to find a roundabout way to access them. In our testing, we tried both ways. Some of our test classes completely ignored private methods, resulting in very low coverage. Other test classes involved implementing new getters() and setters() into the tested classes, resulting in very messy, unused code.

### 2. Abundance of static methods

Again from lectures, we learned that we cannot control the behavior of the static methods. In designing our program, we should have avoided that whenever possible. Unfortunately, we did not consider the testability of our program and our code is rife with static methods, decreasing the possible coverage of our tests.

### 3. Elements of our game we weren't sure how to test

Some aspects of our game contained elements we were not sure how to test. For instance, generating GUI elements such as buttons or the game window itself. We could not determine what assertions to make or how testing a graphical interface even works.

### 2.1.3 Findings

When we first generated our coverage report, we noticed that our coverage percentage was extremely low. This was caused by our tests not accessing a lot of the private methods within the classes. To fix this we added more structural tests to ensure that each private helper method is utilized as well as all the branches and conditions. After following the testing pyramid to reach all the branches we began to notice some test cases failing. For example the maze generation failed at certain boundary conditions such as generating a map with a size of 0. We noticed that by creating these targeted test cases greatly helped us in identifying bugs that we would never have discovered while simply playing the game.

To conclude, we found that testing is crucial in Java, particularly when managing intricate systems with numerous interconnected parts. Without testing we overlooked many errors and struggled to fix bugs. With hindsight, starting with test-driven development right off the bat would have helped tremendously. This experience was probably one of the best ways to learn the benefits and uses of testing.