CMPT 276: Project

Phase 3: Testing

Group 24
Armin Hatami
Jacob Yee
Reid Lockhart
Ryan Marwaha

March 28th, 2025

Methodology

Feature to be unit tested	Explanation
Maze Generation	Main class for generating levels. The game requires all passage tiles to be reachable from any other passage tile.
Maze Path Finding	Need to ensure the path-finding algorithm produces the correct and optimal path from the starting and target position. Needs to handle several edge cases, such as valid paths not existing and start/end positions being the same.
Sound	Ensuring audio playback works correctly, handles invalid files or exceptions, and prevents regression in functionality.
Controller	Takes user input and feeds it into the game engine.
UI Elements	The start button should start the game at a randomly generated level, the tutorial button should start the game at a tutorial level, and pressing the menu button while playing should go back to the main menu. During gameplay, the labels should display the correct amounts.
Game Logic	After every tick, each entity should be in a legal position and any interactions between entities should resolve properly. For example, players should move, enemies should move, rewards should disappear and grant points if a player intersects with them, and so on. In sum: Player movement Player movement checks (Is wall, exit, enemy) Enemy Movement Enemy Movement checks (Is player, is enemy) Enemy Pathfinding Rewards Collection Bonus Rewards Collection Bonus Rewards Disappearance Trap Collision Springs Collision Exits Collision if all Rewards Collected Exits Collision if all Rewards not Collected Winning Losing
TileMap	Every tilemap should set the initial conditions of the game correctly. If the level is smaller than 1, it should initialize one of the tutorial levels. If greater than or equal, it should initialize a generated maze.
LoadLevel	Reads data for the integer map from a JSON file.

Interactions

Interacting components	Explanation
Game, TileMap, MazeGenerator, Entities, LoadLevel	TileMap uses MazeGenerator to create progressive mazes with Entities placed inside. TileMap has an interface with which the Game interacts. TileMap uses LoadLevel to load the integer map from a JSON file for game logic.
Everything, Game, GameListener, Sound	Everything acts as the main entry point and controller for the game. Manages the main menu, game panel, transitions between screens, and background music.

Test Coverage

File	Test Case	Feature/Interaction
TileMapTest.java	testLoadLevel	Checks if a level loaded by LoadLevel is what it should be
	TestValidMoveTrue	Player movement check if a valid move
	TestValidMoveFalse	Player movement check if moving into a wall
	TestValidMoveNegX	Player movement check if X is out of bounds
	TestValidMoveBigX	Player movement check if X is out of bounds
	TestValidMoveNegY	Player movement check if Y is out of bounds
	TestValidMoveBigY	Player movement check if Y is out of bounds
	TestIsExitTile	Exit Check test if not exit
	TestIsExitWall	Exit Check test if not exit
	TestIsExitExit	Exit Check test if exit
	TestGenMaze	Basic Maze Generation sanity testing
GameTest.java	TestMakeMap	Tests that the map for a level is the right map
	TestTickNoMove	Checks that if the player does not input anything, they do not move at the start of the level
	TestClear	Tests that level information is cleared
	TestTickMove	Checks that the player moves for each tick if it’s a valid move
	TestTickMoveWall	Checks that when the player moves into a wall they don’t move

	testPlayerEnemyCollision	Checks that a player’s score is reduced below 0 when touching an enemy i.e they lose
	testEnemyMovement	Checks that the enemy moves towards the player
	testInitializeElements	Checks that Labels get properly initialized and don’t throw errors
	testIfNoPath	Checks that the enemy doesn’t move if there is no path to the player
	testBonus	Checks that the Bonus reward can be collected and increases score
	testBonusVanish	Checks that the bonus reward vanishes when the time is up
	testSpring	Checks that the spring bounces the player back to 1, 1
	testLose	Checks that when the score is negative the player loses
	testReward	Checks that the reward both disappears and increases score when player intersects it
	testWin	Checks that the player wins when the level exceeds 5
	testExit	Checks that the exit doesn’t work when you don’t have all the rewards and that it resets your score when going from tutorial -> normal
	testExitNonReset	Checks that the exit doesn’t work when you don’t have all the rewards and that it doesn’t resets your score when going from normal -> normal
MazeGeneratorTest.java	test1	Generates 1000 random mazes with width/height set to 17/30, asserts that the percentage of walls and passages to be in the 30% to 70% range
	test2	Checks if IllegalArgumentException is thrown when MazeGenerator attempts to generate a 0x0 maze
	test3	Generates a 2x2 maze (only walls)
	test4	Generates a 1x10 maze (only walls)
MazePathFinderTest.java	findPathBFS1	Checks the path returned by the pathfinding algorithm against a known optimal path
	findPathBFS2	Checks the path returned by the pathfinding algorithm against a known optimal path with walls in between
	findPathBFS3	Checks that when the start and end are the same, the pathfinding algorithm returns an empty list
	findPathBFS4	Checks that when there is no path from the start to the target position, the algorithm returns an empty list

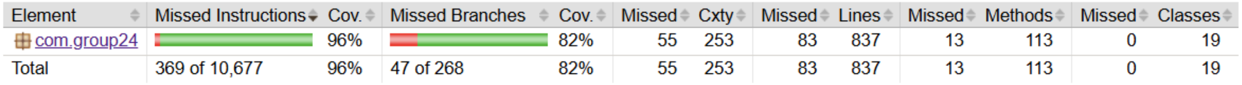
MainUITest.java	MainUITest	Checks to make sure that we can open the menu without throwing errors
	testPlayButton	Checks that the play button starts a randomly generated level
	testTutorialButton	Checks that the tutorial button starts a defined tutorial level
SoundTest.java	test1	Check the standard procedure of playing a sound file
	test2	Checks if an exception is thrown for missing files, bad urls/files
ControllerTest.java	keyPressed	Checks the controller keys to see if they match the encoder output

Test Quality and Coverage
Discuss the measures you took for ensuring the quality of your test cases in your report.

By planning out what functional tests we would need to write before actually writing them, we reduced the complexity and increased the coverage of our tests. For more simple features and interactions, we wrote up domain matrices for them (i.e. Check players entering exit and don't have all rewards, have all rewards, have all rewards and bonus, not have all rewards but have bonus) to ensure that we covered all possible cases. Then, by defining partitions for our project, we were able to divide up the work among us and reduce the duplication of test cases. However, in practice, everyone worked on just about everything.

In addition, using line and branch coverage analysis tools for structural testing allowed us to make sure that no stone was metaphorically left unturned. We also have a couple of comprehensive tests that were generated randomly to catch any other possible errors that we might have overseen.

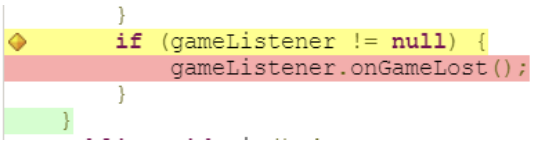
Measure, report, and discuss line and branch coverage of your tests. Document and explain the results in your report. Discuss whether there are any features or code segments that are not covered and why.

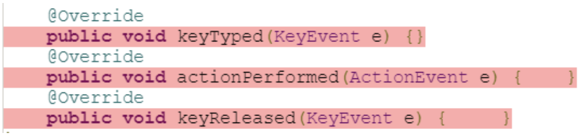


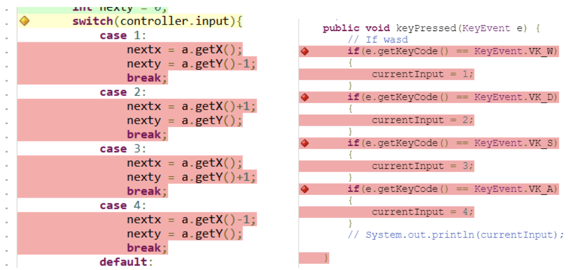
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.group24		96%		82%	55	253	83	837	13	113	0	19
Total	369 of 10,677	96%	47 of 268	82%	55	253	83	837	13	113	0	19

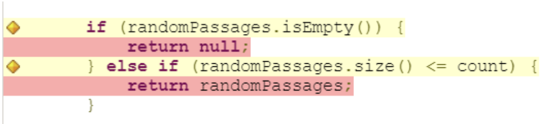
Measuring our test results with Jacoco, we achieved 96% line coverage and 82% branch coverage. This can be verified by running "mvn clean test" and navigating to the index.html file located in target/site/jacoco. This is adequate as by checking the uncovered branches and lines of code on a case by case basis, we find that there are valid reasons to not have them covered by automated unit tests.

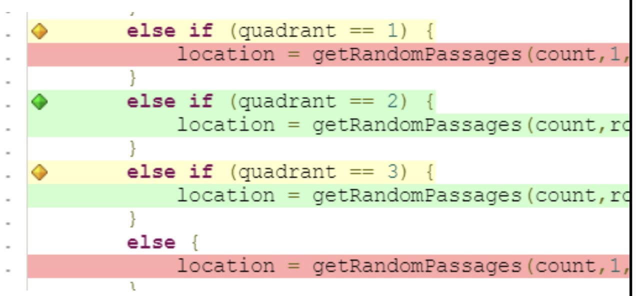
File	Feature/Segment not Covered	Reason

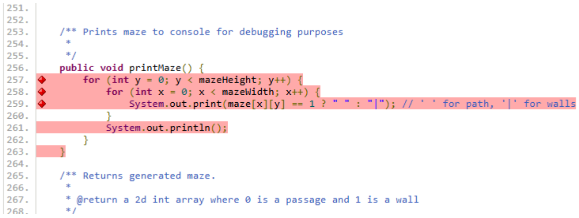
Game.java	
	`}`
	`if (gameListener != null) {`
	`gameListener.onGameLost();`
	`}`
	`}`
		Because we are simulating the game in a headless mode, we don't call the GameListener to send information back to the main JFrame

Game.java	
	`@Override`
	`public void keyTyped(KeyEvent e) {}`
	`@Override`
	`public void actionPerformed(ActionEvent e) { }`
	`@Override`
	`public void keyReleased(KeyEvent e) { }`
		We have simulated keypresses in other ways by using a setter in our tests

Game.java	
	`int nexty = 0;`
	`switch(controller.input){`
	`case 1:`
	`nextx = a.getX();`
	`nexty = a.getY()-1;`
	`break;`
	`case 2:`
	`nextx = a.getX()+1;`
	`nexty = a.getY();`
	`break;`
	`case 3:`
	`nextx = a.getX();`
	`nexty = a.getY()+1;`
	`break;`
	`case 4:`
	`nextx = a.getX()-1;`
	`nexty = a.getY();`
	`break;`
	`default:`
	
	`public void keyPressed(KeyEvent e) {`
	`// If wasd`
	`if(e.getKeyCode() == KeyEvent.VK_W)`
	`{`
	`currentInput = 1;`
	`}`
	`if(e.getKeyCode() == KeyEvent.VK_D)`
	`{`
	`currentInput = 2;`
	`}`
	`if(e.getKeyCode() == KeyEvent.VK_S)`
	`{`
	`currentInput = 3;`
	`}`
	`if(e.getKeyCode() == KeyEvent.VK_A)`
	`{`
	`currentInput = 4;`
	`}`
	`// System.out.println(currentInput);`
	`}`
		We have simulated keypresses in other ways by using a setter in our tests to set the currentInput

MazeGenerator.java	
	`if (randomPassages.isEmpty()) {`
	`return null;`
	`} else if (randomPassages.size() <= count) {`
	`return randomPassages;`
	`}`
		This is simply fallback code that will never be run. Random passages will always exist and never be empty

MazeGenerator.java	
	`else if (quadrant == 1) {`
	`location = getRandomPassages(count,1,`
	`}`
	`else if (quadrant == 2) {`
	`location = getRandomPassages(count,ro`
	`}`
	`else if (quadrant == 3) {`
	`location = getRandomPassages(count,ro`
	`}`
	`else {`
	`location = getRandomPassages(count,1,`
	`}`
		This is simply fallback code that will never be run. Random passages are never located in these quadrants.

MazeGenerator.java	
	`/** Prints maze to console for debugging purposes`
	`*`
	`*/`
	`public void printMaze() {`
	`for (int y = 0; y < mazeHeight; y++) {`
	`for (int x = 0; x < mazeWidth; x++) {`
	`System.out.print(maze[x][y] == 1 ? " " : "|"); // ' ' for path, '|' for walls`
	`}`
	`System.out.println();`
	`}`
	`}`
	
	`/** Returns generated maze.`
	`*`
	`* @return a 2d int array where 0 is a passage and 1 is a wall`
	`*/`
		This method was used for debugging purposes and is kept solely for legacy purposes when refactoring as a sanity check.

Findings

We discovered several bugs and unintended features that we needed to fix by testing our code. In general, creating unit tests with a test-driven development mindset helped uncover missing edge cases from our implementation. An example of this was attempting to generate a 0x0 maze.

In certain situations, the number of rewards needed to proceed onto the next level was incorrect, leading to the player being able to proceed to the next level without the required number of rewards or being locked out of passing the level no matter what they did. It was a minor off-by-one error, and we changed the code to correct the bug. Another error we found from running the tests was a complete crash under very narrow conditions. If a bonus reward was created in a generated maze, and only a generated maze, the entire game would crash due to a malformed file path. We had not discovered this in our manual testing because the situation that it occurred in was quite rare, but it was found in our tests. We fixed it by simply correcting the file path. Another bug was that the player's score did not reset when they finished the tutorial. Completing the last tutorial level immediately puts you into the main game, but the player keeps points for rewards collected during the tutorial. This was very simple to fix once it was noticed.

An additional feature that we added was a win condition. Before this phase, our game kept going infinitely, with the player's goal to maximize their score and see how many levels they could pass. However, it would be more exciting if the player could actually win, so now, passing a certain number of levels will trigger a congratulatory win message displaying the player's final score.

In addition, we refactored our code several times as part of HW4 during this phase and broke it more often than not. The automated testing suite we had created as part of phase 3 was invaluable in acting as an early alert system for when a refactor changed behavior.

In sum, writing and running these tests showed us the importance of automated repeatable testing, especially once we started making more significant changes when refactoring our code as part of HW4.