CMPT 276 Group 18 Phase 3 Report

Rodrigo Arce Diaz, Caleb Hengeveld, Alicia Lam, Shiqi Zhong

Unit and Integration Tests

This project required lots of Unit and integration testing due to its large scale nature. There were plenty of features that needed to be isolated and unit tested. For our testing, we used JUnit test cases to try and cover as many features as possible, and the JaCoco Maven plugin to generate test coverage data.

Boards:

BoardTest.java

Testing the board requires using a headless application. Before every test we have a setup phase which creates a mock Board object, a mock SpriteBatch object, and a mock Skin object. The first 4 tests consisted of checking that the expected number of rewards, punishments, bonus rewards, and bonus punishments were initially set in the construction of the board.Some of the important tests are checking specific blocks on the board such as the starting block and the ending block. These have special properties as it is where the main character starts and ends. We made sure the board was 23 blocks in width and height as well as made sure no blocks lay outside the game area. More tests were used to test the collection objects placed on the board as well as their expected interactions with the character. The last test for board is the draw method which ensures that the draw method does not crash as well as draw bonus objects when they are supposed to.

BlockTest.java

The block test class tests the different kinds of blocks that are used in board creation. For the setup we create a mock texture region and a mock spritebatch. The first three tests are used to test if the different block types can be entered or not. Entering a room block should return true, and the wall and barrier block should return false. The next test checks the position of any given block by getting its x position and y position. We then set a different x and y, and check its position again.

Entities:

CharacterTest.java

The character setup requires creating a mock Board, a mock Block, a mock Spritebatch, and a mock texture region. After that, we create a character with a mock texture region and initial starting point x=1,y=1. The character test class covers a variety of tests that are key features of the character in the main game. These tests check different possibilities of the characters position when initializing the game, or moving throughout the game. In cases where the character is unable to move, we also make sure to test for that. Other such tests check the direction the character is facing when moving. We check the speed of the character, although this is constant, we still check to make sure it is what is expected. We then check if the character can draw while facing different directions as well as without offset. These are all important features that happen for the character when creating the game.

EnemiesTest.java

Much like the character setup, the enemiesTest file requires us to create a mockBoard, mockBlock, mockSpritebatch, and mockTextureRegion. These are initialized in the setup and we test the same kind of movements as we do in character except it's for the enemy. Those being Up,down, left, right, unable to move, as well as direction facing. Although a bad input should never happen, we test out what would happen if a bad input was passed through direction. We then check if an enemy can leave the confines of the world, which fails. The last tests are the same drawing tests we had for character.

Moving_EnemiesTest.java

The setup for the moving enemies test requires the same mock set up as character and enemies as well as creating 4 mock blocks for the different possible walls that a moving character can interact with. These tests cover the different possible aspects of a moving enemy and its decisions depending on where it can move. It takes into account where the enemy is and where the player is.

PatrollingEnemiesTest.java

Before the test, we set up the mock Board, Block, Batch, and TextureRegion. The test uses every method in PatrollingEnemies, starting by checking the initial position, and maximum and minimum x and y-coordinates. It then tests create_path to see if it will change direction when it moves to the end of its path in both the x and y-directions, and uses the mocked Block with the direction method to see if the direction method moves the patrolling enemy correctly.

Punishments:

PunishmentTest.java

The punishment test is a test for two types of punishments: normal punishment and bonus punishment. Before testing, we set up the mock TextureRegion and mock SpriteBatch. We use the mock TextureRegion to create the normal punishment and bonus punishment with score, position. The bonus punishment has additional information to appear for a set time. We first test the get function to check if it can get correct information, and then use the set function to change the information for the punishment as well as use get to check if it correctly sets the information. We use mock Spritebatch and mock Textureregion to test whether the draw function for the punishment can draw the correct size and texture for the punishment.

Rewards:

RewardTest.java

The reward test tests two types of rewards, the regular reward and the bonus reward. Before testing, we set up the mock Textureregion and mock Spritebatch. The regular reward is created by mock texture, position, and score. The bonus reward is created by mock texture, position, score, and a set appearance time. We use the get function in the reward to find out if the setup is correct or not. We use the set function in the reward to change the information for the reward, and then use get to test if it works correctly. Ultimately we use mock Textureregion and mock Spritebatch test draw function for the reward to check if it can correctly draw the size and texture of the reward.

Screens:

MainMenuTest.java, HelpScreenTest.java, GameScreenTest.java, and EndScreenTest.java

The screen tests consisted of checking that the correct screen was set and rendered. Because of the framework we used, we had to implement a TestGame object that would take in a Runnable object to run our test while the application was open. This test ensured that there were no errors in creating the screen and that the respective UI components were generated. Separate screens were tested using different screen classes to ensure total functionality of each screen.

The main menu screen was relatively simple to test as we only had to test that it is a MainMenuScreen object. The same can be said for the test menu screen, as we only had to test that it is a TestMenuScreen object. These same conditions also apply to EndScreen, as we ensure that it is an EndScreen object.

As our Gamescreen contains a lot of moving parts, we had to make more thorough tests to ensure its proper implementation. First we made a runnable that checks if the current screen is a GameScreen. We tried implementing a robot to simulate keystrokes during our test game , however as the tests stand now, it can only make sure the proper Screen subclass is being set. These tests are also done for the Ready screen, and the Pause screen.

Integration:

CharacterCollisionEnemiesTest.java

The GameLogic and GameScreen classes interact with the Character and Enemies classes when GameScreen's GameLogic object calls GameLogic's checkPlayerCollision method. The method call passes in the Character object and an ArrayList containing all of the enemies on the board, and accesses all of their x and y coordinates to compare every enemy's position to the player's. If any enemy has the same position as the player, it returns true in GameScreen, which then calls playerEnd(false) to move to the playerloses version of EndScreen. CharacterCollisionEnemiesTest has two tests- one using moving enemies and the other using patrolling enemies- which both instantiate a Character, an ArrayList<Enemies>, and several enemies at different locations which are added one at a time. In between each addition to the arraylist, we test if checkPlayerCollision returns false; the last enemy added has the same position as the Character, so we test if checkPlayerCollision returns true.

CharacterCollisionBlockTest.java

The Character class interacts with various subclasses of the Block class (RoomBlock, BarrierBlock, Wall) when it attempts to move from one block to the other. The Character's direction method gets a block from a mocked Board and calls the block's enter method, which returns true or false; if true, the Character's position is modified accordingly. CharacterCollisionWallsTest has four different tests for each Block subclass, which attempt to move the player up, down, left or right, and checks the Character's position afterward to see if it behaves correctly.

CollectPunishmentsTest.java

The GameLogic and GameScreen classes interact with the Character and Punishment classes when GameScreen's GameLogic object calls GameLogic's checkPunishment method. The method call passes in the Character, the Board, and GameScreen's time, gets the Character's position, and sets the amount to decrease the score to a sum of the board's regPunishmentCollect and bonPunishmentCollect method. Both methods call another method to determine if a regular or bonus punishment respectively is in the same block as the Character, and to get and return the index of the punishment in its array. If the index exists, it gets and returns the punishment's value, and removes the punishment from the array. getPunishment uses the punishment value to decrease the player's score. CollectPunishmentsTest tests the game's behaviour when collecting a regular or bonus punishment, and when collecting both at once, and checks the player's score afterwards to make sure it is correct.

CollectRewardsTest.java

The GameLogic and GameScreen classes interact with the Character and Reward classes when GameScreen's GameLogic object calls GameLogic's checkReward method, which works much like checkPunishment, except it works with Rewards instead of Punishments. CollectRewardsTest tests the game's behaviour when collecting a regular or bonus reward, and when collecting both at once, and checks the player's score afterwards to make sure it is correct.

EndGameTest.java

The GameLogic and GameScreen classes interact with the Character class when GameScreen's GameLogic object calls GameLogic's checkScore method. The method call passes in the Character, gets the Character's score, and checks if their score is greater than or equal to 0; if it is, it returns true in GameScreen, which calls playerEnd(false) to move to the playerloses version of EndScreen. Additionally, the GameLogic object calls its checkIfExitingMaze method in GameScreen. The method call passes in the Character and Board, gets the Character and the End Block's position, and gets the number of regular rewards remaining on the board. If there are no more regular rewards and the Character's position matches the End Block's position, it returns true to GameScreen, which calls playerEnd(true) to move to the playerwins version of EndScreen. EndGameTest has one test that changes a Character object's score and makes sure checkScore returns true or false accordingly, and another test that changes the Character and EndBlock position, and the number of regular rewards remaining, and makes sure checkIfExitingMaze returns true or false accordingly.

EnemiesMoveTest.java

This test is GameLogic interaction with the Enemies class(Patrolling Enemies, Moving Enemies)and the block class(Wall class, RoomBlock Class, BarrierBlock class). Before the test, we set up the mock Character in a given position. In the test it give out the arraylist for the two types of enemies and the arraylist for boolean with start all false. We use the method moveEnemies in GameLogic class to check if the enemies move when they have different block classes.

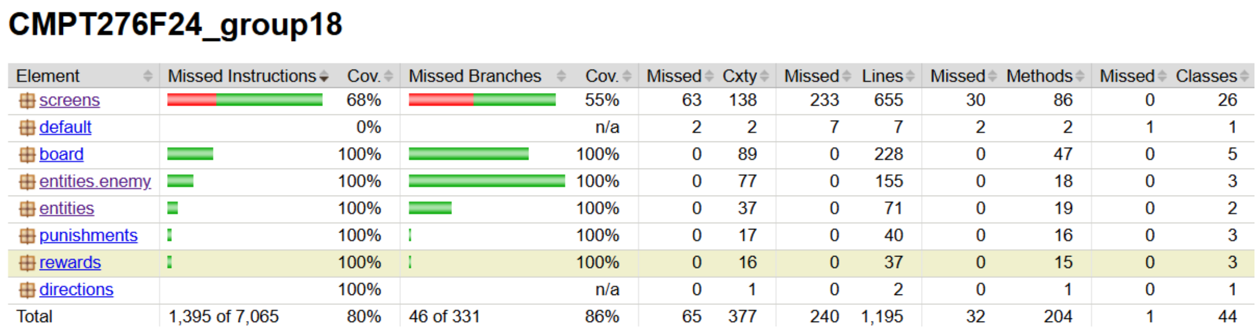
MovingEnemiesCollisionBlockTest.java and PatrollingEnemiesCollisionBlockTest.java

These two tests are similar. We test the interaction with Block class(Wall class, RoomBlock class, BarrierBlock) and Enemies class(Patrolling Enemies, Moving Enemies). The test checks when the enemies move in a direction, by using enemies.direction for the two types of enemies to check whether the movement of the two types of enemies is correct or not. This test checks for four directions which are up, down, left, right.

Test Quality and Coverage

When creating test cases for our classes, we generally created functional test cases first, before looking at what wasn't being tested and creating structural tests to test the remainder. To avoid repeated code, we created setup methods which executed before every test in its class, and cleanup methods that executed after every test class where needed.

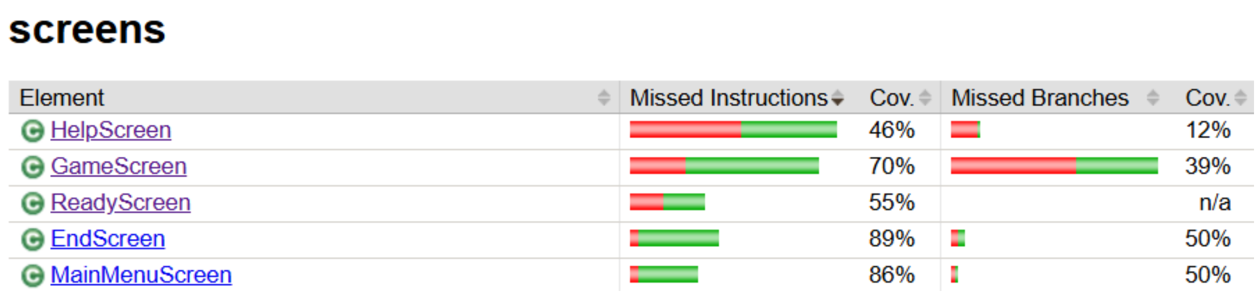
We ran 129 tests with no failures, errors, or skipped tests, and got 79% instruction coverage and 86% branch coverage.



CMPT276F24_group18

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
screens		68%		55%	63	138	233	655	30	86	0	26
default		0%		n/a	2	2	7	7	2	2	1	1
board		100%		100%	0	89	0	228	0	47	0	5
entities.enemy		100%		100%	0	77	0	155	0	18	0	3
entities		100%		100%	0	37	0	71	0	19	0	2
punishments		100%		100%	0	17	0	40	0	16	0	3
rewards		100%		100%	0	16	0	37	0	15	0	3
directions		100%		n/a	0	1	0	2	0	1	0	1
Total	1,395 of 7,065	80%	46 of 331	86%	65	377	240	1,195	32	204	1	44

All of our missed instructions and branches were in the screens package, specifically in MainMenuScreen, HelpScreen, ReadyScreen, GameScreen, and EndScreen.



screens

Element	Missed Instructions	Cov.	Missed Branches	Cov.
HelpScreen	🟥🟩	46%	🟥	12%
GameScreen	🟥🟩	70%	🟥🟩	39%
ReadyScreen	🟥🟩	55%		n/a
EndScreen	🟥🟩	89%	🟥🟩	50%
MainMenuScreen	🟥🟩	86%	🟥🟩	50%

Our test coverage was primarily limited by our inability to simulate mouse clicks or keyboard input, which prevented us from accessing functions triggered by user input, such as the ChangeListeners for buttons, or the draw methods for moving entities which implemented an offset to make their movement render smoothly.

Findings

During this phase, we had to make many changes to our base code in order to make it more testable. Originally, in the creation of each new screen, we would instantiate SpriteBatch, Skin, Font, and Texture objects which remained the same throughout all of our screens; we refactored our code so that they were created once in MazeGame and passed into each new screen. Additionally, the screen classes' constructors were cluttered with button and event handler instantiations, so we reorganized and moved said code into the private method createButtons.

GameScreen was behaving as a god class, so we extracted the game logic into GameLogic.java to limit GameScreen's responsibilities, while also making integration tests for object collision, point collection, and endgame conditions easier to test and alter. Additionally, the Ready screen showing the full board prior to starting the game was extracted into its own class and the pause screen was put into an inner class. This made the long list of parameters in the initial GameScreen class significantly smaller.