CMPT 276: Group Project

Spring 2020

Phase 3- Group #9

Introduction

For our testing phase of the project, we tried to implement both techniques of automated testing and manual testing. For automated testing, as the assignment stated, we implemented components of black-box testing which included unit-testing as well as integration testing. We used Maven and JUnit to compute all of our tests and we documented all of our tests in the table below.

Unit Testing

Unit Tests

S. No	Action	Inputs	Expected Output	Actual Output	Test Result
Player					
PU_1	Test the players position methods (setters and getters)	int x and int y	100, 100	100, 100	Pass
PU_2, PU_3	Testing the players life system by checking if the player is out of lives	void	False, True	False, True	Pass
PU_4, PU_5, PU_6, PU_7	Testing the players score system by using the addScore, decreaseScore, and resetScore methods	Int score (the score to be addded/decreased)	0, 100, 50, 0	0, 100, 50, 0	Pass
PU_8, PU_9	Testing the players movement by setting the setUp, setDown, setRight, and setLeft methods then calling update() to advance the players position one cycle	Boolean b (true/false)	The players position: After moving down: 102 After moving right: 102 After moving up: 100 After moving left: 100	102, 102, 100, 100	Pass
Reward					
RU_1 RU_2	Testing the player's score increased by reaching the reward and bonus reward,	Int score(the score to be increased)	Player reach reward and get: 1 Player reach bonus reward and get: 1	1 1	Pass

Map					
MU_1 MU_2	Testing the map's ability in detecting walls Testing if the map is recognizes not a wall	Int x and int y Int x and int y	True False	T1. Null Pointer exception T2. True False	Error Passed
Enemy					
EU_1	Test the enemies position(setters and getters)	Int x and int y	4, 8	4, 8	Passed
EU_2	Testing correct respawn positions	void	4, 8	4, 8	Passed
ExitPoint					
EPU_1	Testing if the exit point is correctly spawning	Void	4,8	4,8	Passed
EPU_2	Testing if the exit point becomes non-active	Void	False	False	Passed
EPU_3	Testing if the exit point becomes set to active	Void	True	True	Passed
EPU_4	Testing if the exit point is returning the correct coordinates	Void	4, 2	4, 2	Passed
GameTimerTest					
GMU_1	Testing that the Timer is initialized	void	Timer initialized	Timer initialized	Passed

GMU_2	Testing that the time given in seconds making	void	Returns 0	Returns 0	passed
GMU_3	Testing that the time is not negative	void	True	True	Passed
GMU_4	Testing that the timer is working and is not null	void	Return not null value, timer created	Returns not null value	Passed
Door					
D_U1	Testing that door is closed	void	False	False	Passed
D_U2	Testing that trap is setup	void	True	True	Passed
D_U3	Testing that whether there has a trap	void	False	False	Passed
D_U4	Testing after door respawn and it is closed	void	True	True	Passed
D_U5	Testing if the door isclosed after door respawn	void	True	True	Passed
D_U6	Testing if the door location is correct	void	x=10 y=10	x=10 y=10	Passed
D_U7	Testing if the ID is correct	void	0	0	Passed

Figure 1 - A record of all the Unit tests that was done (*full pdf on gitlab*)

For the player, we tested all the individual components that made the player or controlled the things the player could do such as move. We ran a series of tests on the player's position, the player's lives, the player's score, and most importantly the player's movement. To test the player's position functions we set the position and checked if the returned value was equal to the set position. We tested the players respawn position with the same techniques as well. We tested the player's live system by checking whether the player was alive or not by asserting true and false. If the player was alive, we asserted true. We then reduced the player's life using the method we use to calculate damage and then asserted false if the player was alive after losing all of their life. We tested the score by adjusting its values with the increase and decrease score methods then verified the behavior was correct after the values had been adjusted. Finally, we tested the player's movement by testing all directions and seeing if the player's position changed appropriately. All tests were successful and no issues were found. Everything was behaving as we intended.

For all of the remaining classes to be tested we followed the same testing philosophy by testing each method of the class. This philosophy included testing every possible scenario that could happen with these units and was further tested when we ran our integration tests.

Integration Testing

Integration Tests

S.no	Action	Inputs	Expected Output	Actual Output	Test result
Player					
P_I1	Player touches reward	Reward	Reward collected Score increase Inventory increase	The score increased	Passed
P_I2	Player touches bonus reward	BonusReward	Bonus reward collected Score increase	The score increased	Passed
P_I3	Player touches trap	Trap	Trap Collected Decrease score	The score decreased	Passed
P_I4	Player touches trap from above	Player move from above	Trap trigger Trap collected and decrease score, player lost life	Trap trigger Trap collected and decrease score, player lost life	Passed
P_I5	Player touches trap from below	Player move from below	Trap trigger Trap collected and decrease score, player lost life	Trap trigger Trap collected and decrease score, player lost life	Passed
P_I6	Player touches trap from left	Player move from left	Trap trigger Trap collected and decrease score, player lost life	Trap trigger Trap collected and decrease score, player lost life	Passed
P_I7	Player touches trap from right	Player move from right	Trap trigger Trap collected and decrease score, player	Trap trigger Trap collected and decrease score, player	Passed
			lost life	lost life	
P_I8	Test that the player detects the door from below, test if not from below above, test if not from above right, test if not from right left, test if not from left	Door	Return True Return False Return True Return False Return True Return False Return True Return False	Return True Return False Return True Return False Return True Return False Return True Return False	Passed
P_I9	Test the players interaction with the ExitPoint from all directions (up, down, left, right)	ExitPoint	From above: true From below: true From right: True From left: true Not entering it: false	True, true, true, true, false	Passed
Enemy					
  E_I1 E_I2 E_I3	Testing Enemy Hit Player so Player position respawns Enemy position respawns and Player decrease life	Player	 4,4 200, 201 2	 4,4 200,201 2	Passed
E_I4 E_I5 E_I6	Test that the enemy detects the door from below, test if not from below above, test if not from above right, test if not from right	Door	Return True Return False Return True Return False Return True Return False	Return True Return False Return True Return False Return True Return False	Passed

E_I7	left, test if not from left		Return True Return False	Return True Return False	
E_I8	Test Enemy moving when player moving	Player	X != 4 Y != 8 X != 4 Y != 8	T1. Null Pointer Exception T2. Enemy moves	Failed Passed
E_I9	Test The enemy not moving when player is moving	Player	4,8 4,8	4,8 4,8	Passed
MenuState					
   MS_1 MS_2 MS_3 MS_4 MS_5 MS_6	Testing up and down buttons to change with text is highlighted in menu states Enter Down Down Down Up Up	Game Manager	Gm.currentstate should be specified state 0 1 2 0 2 1	 0 1 2 0 2 1	Passed
Applies to all other game state classes in the main menu screens					

Figure 2 - A record of all the integration tests that was done (*full pdf on Gitlab*)

For the integration tests displayed in **Figure 2**, we found the best way for testing our game was to test all the possible interactions that could happen in the game. These interactions were basically limited to how the player object interacts with the other objects. So we broke down all the possible scenarios we could think of into a table and split the work up evenly. For the majority of the tests we tested how the player interacts with the object just from a stand still and then we also tested how the player interacts with the object from every possible direction. We did this to ensure that the player couldn't walk through an item or trap and have nothing happen. For example, say the player walks through an item and they collect it but the item remains and the player has a way of infinitely gaining points. By testing all directions and making sure the item behaves properly when interacting with the player we can prevent the interaction from misbehaving. In conjunction with these integration tests we also did some black box testing to get a visual confirmation that everything behaved properly once again.

Like the example with the item, all of our integration tests followed this practice and with the help of it we discovered a few bugs such as the exit point not refreshing after the player has opened it. During our blackbox testing we found that if the player collects enough items to open the exit point but then game overs and restarts the level, the exit point would remain open. The reason for this was we didn't create a method to reset the players inventory when they have a game over and this probably would have been completely overlooked without the testing phase.

Features or Code segments not covered

For Unit and Integration testing, we mainly focused on the game objects given a certain scenario. This means that we avoided any features of a game object that ran over a runnable thread. In the testing code, we found that it was difficult to assert a certain outcome for runnable objects that can potentially change over time. A good example of a feature or code segment that represents the problem would be checking if the game timer is incrementing correctly or not. As the game timer was run on a separate thread that could only work if the program is constantly running, considering that the test code runs and computes once and does not run long enough to see if the timer is incrementing time properly, we were not able to write a test code that determines if the timer was working. Our alternative to this scenario was to simply check if there is at least an instantiation of this timer task when the game timer is instantiated. Other aspects that were not tested include code segments and classes that did not contribute to the logistics and functionality of our game. These aspects include the code that handles the graphics or the game environment such as the Animation class, Tile class and many components of the map class since they all either determined the layout of the game environment or they were just aesthetics of the game rather than actual game logic. For the game states, we only wrote test cases for only the menu state that tested if the menu was properly toggling the texts correctly on button commands. This is because, the other game states, such as the Win state or How-to-play state mimic the design of the menu state and have the exact functionality. That is why, it is safe that if one of them works correctly, all of them should portray behaviour respectively. Similarly, we also did not write any tests for the game panel or the game state manager because the game panel only acts as the canvas that determines where the game takes place and the game state manager handles which state is selected to appear in the JFrame. Neither has any importance in our game logic or interacts with the user directly.

What we learned

Throughout phase 2 of our project, we already fixed a series of bugs that we discovered through sheer manual testing and playing the game. However, we did indeed learn a reasonable amount of the functionality of our game through automated testing. We definitely learned the importance of automated testing and how it can help assure the quality and functionality of our code. Firstly, in order to test a lot of characteristics of game objects, we realized that we have forgotten several getter functions to access the state of an object. For example, we included the getPlayerLives function to see the current state of the player's life, just in case we needed to require the number of lives of a player for our future implementations. Moreover, when we set the player's coordinate for the test, it was hard to find the accurate matching position, so we used the loop function to make the player move one step at a time to match the position. Also inspecting our code to write tests for them, we also discovered some functions that did not serve

any purpose to our game so we decided to ultimately delete them. But other than reorienting some of our classes around to fit our system better, we did not make any drastic changes to the core layout of our game. Moreover, when we execute the test, we can break the code into very small pieces and test the class individually, and it is very helpful to identify the bug in an efficient way because in the future, if we are coding a huge program it is not easy to debug without doing the test.

Important Findings

Some important findings that came out of this automated test process were a unique bug in the exit point class and instantiation. The exit point is supposed to be a spot in the map that only becomes active whenever a specific condition is satisfied, for example, the player has collected five rewards, but we discovered through this testing phase, that the exit point remains open even after the level is reset. Therefore, allowing the player to walk right to the exit point and win the game before they even collect any rewards. This caused us to make the necessary adjustments to the exit point class. Also, we also learned that a map must foremost load the tiles for the map to be accessible and allow game objects to be placed on top of the map, otherwise, the test will return a null pointer exception.

Overall, through the process of phase 3 that allowed us to experience automated testing for the first time. We have definitely learned to not undervalue the importance of automated testing, considering that testing is a key component in assuring the quality of code. Lastly, through this phase of the project, we are grateful to learn these several testing techniques being black-box testing using the unit and integration testing and we hope to be able to use these techniques in future projects as well.