



Phase 3 Report

Game Logic Interactions:

Given the nature of how we stored data in a 2D array of Objects which were simply an enumeration of elements, we could do most of the logic testing through the array. However, enemies, hero, rewards, traps, and exit door are objects, and testing helps solidify that both the board and the objects instances were both getting updated correctly.

Unit test: Enemy Movement -> **Intergration Test:** Enemy Collision with rewards /bonus /traps/ player /other enemies -> check game is won/lost,

Unit test: Player Movement -> **Intergration Test:** Player Collision with rewards/bonus/traps/walls/winning by exit

Test

Interactions	Unit tests	Integration tests
Hero Movement on valid keypress and process interaction of hero and tile in all directions (UP,DOWN,LEFT,RIGHT)	Hero can move on a tile Hero position gets changed ProcessPlayerMovement successfully gets called Hero stepping over bush	Tests for each difficulty: EASY, MEDIUM, HARD Hero collision with rewards /bonus /traps/ player /other enemies -> triggers game over 2D array + object attributes such as position gets updated Hero on bush changes isHidden status Hero exiting door and setting game to win Hero exiting door and no movements (not win condition) Hero getting negative points sets the game to loss Hero picking up all rewards changes exit closed status to open Hero gets to trees and no movement Hero gets to bush Exiting door triggers win condition
Enemy Movement	Enemy moving onto static entity tile has have both objects on tile	Enemy paths towards the hero in open space Enemy paths towards hero when obstruction in front

		<p>Enemy paths towards hero if hero is hidden and enemy is within 3 tiles</p> <p>Enemy cannot path toward hero if hero is hidden and if enemy is too far away -Hero paths randomly and does not collide with a tree</p> <p>Enemy must wait if next movement is another enemy on the same tile</p> <p>Enemy Colliding with player triggers game over condition to true</p>
Enemy count	<p>Num of Generated Traps</p> <p>Num of Generated Enemies</p>	<p>Test for each difficulty: EASY, MEDIUM, HARD</p> <p>Count the number of traps and moving enemies for each difficulty.</p>
Map Generation	<p>-All entities get generated</p> <p>-Different difficulties generate the same number of objects for each category</p> <p>-2D array gets initialized</p>	<p>Each map is unique to roughly 2/3 of all other map generations for all difficulties</p> <p>Map does not generate loops where enemies/players cannot get into (map is open by definition)</p> <p>Generating 10000 maps does not run into any timeout errors</p> <p>Enemies get generated a certain distance away from players' initial spawn</p>
Clicking buttons to switch game displays	Check door closed condition	<p>MyDifficulty buttons change the games difficulty state + generate a board with the correct</p> <p>Mysettings unmute mute button change the sound</p> <p>Mysettings leftscroll and rightscroll changes the squirrel's color</p> <p>Pause Menu gets generated successfully</p>
Loading resource PNGs	Loading resource PNGs from resources folder	Display gets rendered based on photos that were selected from resources folder
Key presses	Key presses	Keyevent triggers pressed key and resets keys when necessary

PlayerMovement

Testing game logic needed to ensure that all hero movements and interactions correctly updated the board where the interaction happened, and both objects for which were affected (hero + reward/trap)

Unit test for player movement:

First, testing non-constrained hero movement was critical in the first layer of testing. Tests generated a board which then put the hero into a position where all adjacent tiles were set to empty tiles. In each direction for the movement, called `ProcessPlayerMovement` from `herologic`, the test asserted that the moved-to position now had object `HERO` in the array, and the previous position on the array was an `EMPTY` object. Moreover, the tests below asserted that the hero object instance's position attribute was correctly updated.

Once tests covered movement, we then ran the same tests except adjacent tiles were set to objects different from empty tiles. Tests asserted that picking up objects updated the hero's score and that the 2D array no longer contained a bonus/reward/trap after stepping back to the initial position. Note, tests needed to be written for every difficulty level because traps/rewards changed the hero's score differently for each difficulty. We do this by passing difficulty to the initial setting and specifying the expected reward/trap value based on difficulty.

Note: Bonus Rewards require different types of testing because they have a property of respawning. As a result, when a player picked them up, test cases needed to assert that the objects `isSpawned` variable is set to false.

For movements where obstruction did not let hero move, such as for `TREE` and `EXIT`, tests asserted that player did not move if it was a `TREE`, or if `EXIT` object when attribute closed was true (the case that hero has not collected enough rewards and tries to go to exit). In this case, we had to check three things in order to make sure the test passes.

1. `ExitDoor` is still closed
2. The position of the hero did not change
3. `IsGameWon` attribute is still false (we did not win the game yet)

Note that we need to create different tests for different difficulty levels for movement to `EXIT`. As the number of the collected rewards is different for each difficulty, it might end up in winning the game or no change in character position and not winning the game (5 collected rewards would be a win condition in easy mode and not in medium and hard modes). Here is a table of 6 different tests that were needed for movements to `EXIT`.

Difficulty	closed		Test
EASY	True (collected rewards <5)	No movement for hero	<code>MoveIntoDoorEasyNotWin</code>
MEDIUM	true (collected rewards <10)	No movement for hero	<code>MoveIntoDoorMediumNotWin</code>
HARD	true (collected rewards <15)	No movement for hero	<code>MoveIntoDoorHardNotWin</code>
EASY	false (collected rewards =5)	WIN the game	<code>WinEasy</code>
MEDIUM	false (collected rewards =10)	WIN the game	<code>WinMedium</code>
HARD	false (collected rewards =15)	WIN the game	<code>WinHard</code>

Bonus Rewards require additional assert tests because of their property for despawning when hero picks up the bonus. As a result, when a player picked them up, test cases needed to assert that the objects `isSpawned` variable is set to false, as well as the timer for `despawnedTime` gets updated as well.

Enemy Movement Testing

The logic behind enemy movements was largely comprised of a breadth first search algorithm to find the shortest path to the hero. To test this algorithm a variety of unit tests:

- `UnblockedShortestPathX()` where x belongs to {North, East, South, West}: Asserts that with no blocking trees, the shortest path algorithm moves in the correct direction.
- `BlockedShortestPathBasic()`: a basic map, where direct path is blocked with a tree and an alternate path must be found
- `BlockedShortestPathAdvanced()`: Several blocked paths requiring a convoluted and long path to find arrive at the hero

The prior enemy movement tests were all assuming that the hero was not in a hiding position. When the hero is hiding, two scenarios can occur. First, an enemy is within 3 movements from the hero, in this case the enemy tracks normally (as above). The second, is when the enemy is more than three movements away from the hero, in which case it cannot “see” the hero and will move in a random direction. The following tests assure this behavior:

- `RegularMovementWhenHeroHiddenAndClose()`: as the name describes, asserts that the algorithm returns the direction on the shortest path.
- `RandomMovementWhenHeroHiddenAndFar()`: asserts that the algorithm returns a random direction*

* in this case the random direction is literally the `RANDOM` variant of the `Direction` enumeration.

Since the `getRandomDirection` method is private, to test this behavior a test forces the hero to be hidden and far away from the enemies, and an assertion is made that the enemies location has changed, i.e., they moved. We cannot assert a specific direction as it is randomized.

- `GenerateRandomDirection();`

Finally, now that we know that the enemies will move in the correct fashion for all scenarios, one more test assert that when processing the enemy movements, the enemies do indeed change locations

- `ConfirmEnemyMoved();`

Enemies could be on the same tile as rewards, bonuses, bushes, and traps. By placing an enemy some spaces away from the hero and putting a static entity in front of the enemy, the tests asserted that when an enemy moves once onto the tile containing the entity, the enemy’s previous tile becomes empty, as well as currently being an enemy and entity object on their current tile. After one more movement, the enemy separates from the static entity, and the enemy’s current tile is an enemy, while previous tile turns back to a static entity.

- `EnemyonReward(), EnemyonBonusReward(), EnemyonBush(), EnemyonTrap()`

Enemy Count Testing

When the Play button is pressed, there will be a set number of enemies that will be on the board depending on difficulty selected. Enemies were randomly placed on the board initially, so by iterating through the board which was an array of 2D objects, the test matches the number of enemies found to the number of enemies set for each difficulty.

- `EnemyEasyCount(), EnemyMediumCount(), EnemyHardCount()`
- `TrapEasyCount(), TrapMediumCount(), TrapHardCount()`

Key Handler Tests

Based on the implementation our key handler the test had to run an instance of the game, after which, using Java's Robot class, the "w, a,s,d, esc" keys were pressed and released. Meanwhile assert statements confirmed that the keyhandler changed the state of the respective boolean

- KeyHandlerTest()

Loading Png Resources:

A simple test to assert that all resources can be loaded without an exception getting thrown

- LoadPngResources()

Map Generation Tests:

Due to the Random nature of our map generation, we are unable to assert that specific objects are locations. Instead, we make sure that a correct amount (either exact or lower bounds) of the entity is generated on the board.

- OuterBoardGenerationTest(): asserts that there are barriers around the entire map except for spawn and exit
- HeroGenerationCount(): asserts that there is only a single hero occurrence on the board
- ExitGeneration Count (): asserts that there is only a single exit occurrence on the board
- TreeGenerationCount(): asserts that the tree count is bounded between 1 and max tiles on board
- BushGenerationCount(): asserts that there are 5 bushes generated at each difficulty
- EnemyGenerationCount(): asserts that there are 1,2,3 enemies spawned for easy, med, and hard difficulties
- TrapGenertaionCount(): asserts that there are 4,7,11 traps spawned for easy, med, and hard difficulties
- RewardGenerationCount(): asserts that there are 5,10,15 regular rewards spawned for easy, med, and hard difficulties

Additionally, we have a few tests that make sure that the enemies have a minimum spawn distance from the hero and that rewards have a minimum distance from each other

- MinDistanceBetweenEnemyAndHero(): asserts a min path length of 10, 7, 5 between the hero and enemies for easy, med , and hard difficulties
- MinDistanceBetweenRewards(): asserts a min distance of 5,3,0 between rewards for easy, med, and hard difficulties

However, for wall generation which was entirely random to a point where the number of TREE objects generated is also random, the only way to test consistency was to check that no closed off areas of the map existed. To do so, we ran depth first search starting at position 2Darray[1][1] (test asserted that it was an empty non tree object), and counted every non-tree object the search ran to. If the total numbers of tiles counted equals to the number of total non-tree tiles, then no loops were found. Because the generation was random, the test needed to be run 1000 because there is no possible way to generate every possibility, so a mass generation was the closest approximation to testing the generation.

TestWallGenerationAllDifficulties()

Moreover, random map generation is only random if it is different from other generations. For all difficulties, taking 1 map and comparing it to 100 generated ones, and averaging the percent of tiles that were the same, the test asserts that 35% of the map was similar. Conclusively, every map is roughly 2/3 unique to all other generations of the map across EASY, MEDIUM, HARD.

testMapSimilarity()

Findings

Bug fixes:

After noticing in the first iteration of enemy movement test where each enemy processes their movement one at a time in an order, if the 1st enemy was behind the 2nd enemy, the 1st would not move and the 2nd would move, leaving a gap between them. By updating the code and letting the order of processes switch when encountering this scenario, enemies could have no gaps in-between them.

Resource management:

Testing objects such as JPanels, JButtons, or JFrame often required a lot of resources and caused memory leaks. Although individual tests were passing, and maybe even certain subdirectories were working at a time, however running all tests caused an OutOfMemory error for java heap space. As a result, it was important to remove all internal displays inside the main DisplayLayout object which is required for every test (deep remove). And also call System.gc() after every test using the @AfterEach line before a teardown () function to remove any unnecessary memory that the garbage collection did not remove.

Private variables/methods:

It is necessary to add getters for many of the attributes within classes when testing. For example, when adding tests for picking up rewards, it wasn't enough to simply add a Reward tile onto the 2D array right next to the hero to pick up. It was also required to obtain the private ArrayList<Rewards> attribute from ObjectData instance where a new reward would need to be appended since processPlayerMovement() needs to reference a score attribute from a collection of Reward objects. Consequently, a getter was added here as well as for most of the other classes which required testing.

Private methods were simply called through the public methods in systematic ways to test each control path/case.

Along with programmatic testing, we also performed vigorous manual testing. We made sure all buttons, key presses, and game functionalities worked as expected when running the game. During this testing, we found a major bug in our implementation that did not allow the game to be run on lower resolution screens. Mainly laptops. So, we had to refactor our code to use the screen resolution of the device for rendering everything rather than a predetermined one.

Test Quality/Coverage:

The line coverage of our test suite measured out to be 88.2% with 100% class coverage, 88% method coverage, and 80.3% branch coverage. (please open "index.html" in the Coverage folder for full report). A relatively large portion of our lost branch and line coverage ended up being due to not being able to cover all the cases with the gui (Display package), for example the win screen, after collecting all rewards. Additionally, due to the nature of the random generation of a map, it is much harder to cover all the different possibilities consistently.

For test quality, we made sure to separate functionality into different unit tests and avoid single tests that cover many different aspects. We also utilized different types of Assertions provided by the Junit library to make tests comprehensible and easily diagnosed when failed.