Unit and Integration Test

Introduction

For this report, we define one "unit" as an object group in the game. An "object group" is defined as one of the 8 collection of classes that operates a group of related mechanisms (Audio, Enemy, GameEngine, Map, Overlay, Player, Reward, and Utility). A "unit test" is defined as a test that consists of only details sourcing from a single object group. A test is an "integration test" if and only if the test consists of major participation of details sourced from two or more object groups. A "feature" is a single method within a class and tests evaluate the correctness of these features.

When testing a feature, we will assume all external features are correct and may utilize them for testing the feature in question. Additionally, we are aiming not to test for hardcoded values, but rather for general correctness as these values can change based on the needs of balancing the game; for example, we do not test if the health of the player is exactly "5", but rather if it is non-zero. For each class in the game (i.e. Trap), a corresponding test class (i.e. TrapTest) is made that attempts to test each individual method in that class.

A majority of testable features in the game can be categorized into one of 8 distinct function groups: *getters*, *setters*, *adders*, *checkers*, *spawners*, *movers*, *colliders*, and *resetters*. We will discuss our testing approach for each of these 8 general function groups.

It should be noted that, due to the nature of automated testing, features involving audio and visuals are extremely difficult to test through automation yet extremely trivial when manually tested. Therefore, any testing of auditory or visually related features will be avoided. Additionally, since Java lacks explicit calls to deconstructors, testing for static methods when the class is uninitialized will be largely ignored in our testings.

Getters

Methods in the *getter* function group require no input arguments and are public. Most object groups contain few *getter* methods, and most *getter* methods perform extremely simple tasks such as returning a global variable that is stored in its respective class. Examples of these simple one or two-line methods include getX() of the GameEngine object group, getMaxHealth() of the Player object group, and getElusiveHeal() of the Reward object group. These methods are dependent on only their respective class (and their respective object group) and thus can be easily unit tested. Tests for these values usually revolve around testing for general correctness if defaulted values are present (i.e. x-coordinate position

value must be non-zero); otherwise, we can test the *getters* by matching their corresponding *setter* methods.

Certain features like canBeDamaged() in Hirsch, of the Player object group, contain time dependencies as our game mechanism allows the player a period of invulnerability after receiving damage. However, this can still be easily unit tested via utilizing the sleep function to wait out the period of invulnerability (or fail if the wait exceeds the timeout period).

There is one special instance of *getter* method that resides in GameManager of the GameEngine group: *getGameStatus()*. The feature returns the current game status (i.e. victory or defeat). To test for correctness of this feature, we need to emulate the condition of each game state by manipulating the various object groups. Hence, this feature can only be only with integration testing.

Setters

Methods in the *setter* function group, in general, have one input argument, returns void, and are public. *Setter* methods exist in only two object groups: GameObject of the GameEngine object group, and Hirsch of the Player object group. All *setters* in GameObject are extremely simple one-liner methods that simply alter the value of a stored variable. These features can be trivially unit tested by assessing the correctness of a *getter*'s return value after setting a value via its corresponding *setters*.

For the *setters* in Hirsch, setDeath() is a special case of *setter* function with no input argument. This method simply sets the health of the player to zero, so simply testing for the health value to be zero would be sufficient.

Adders

Methods in the *adder* function group return void and are public. *Adder* methods appear only in the Hirsch class of the Player object group, performing simple arithmetic by incrementing or decrementing existing global variables. Like the *getter* function group above, the *adder* methods are dependent only on their own respective classes and can easily be unit tested via utilization of the *getter* methods. One way to test these methods involves iterating the *adder* methods a random number of times, then using a corresponding *getter* method to check for correctness.

Checkers

Checker functions appear only in the Map class of the Map object group, and PlayerManager class, of the Player object group. For the Map's *checker* function group, isEntrance() and isWall(), the functions are public and are dependent only on its respective

class; this makes it easily unit testable. To test these methods, we will use various known map coordinates and assess for correct boolean return values.

PlayerManager has two private *checker* functions: checkEnemy() and checkReward(). checkEnemy() makes calls to EnemyManager and checkReward() makes calls to the RewardManager. The two *checker* functions obtain collision data and update the game state variables stored in the Player object group accordingly. Due to the nature of these two *checker* methods and the fact they are private, integration tests are needed to evaluate its correctness. To test we need to simulate collisions between player objects and enemy or reward objects and examine the resultant changes in order to verify these *checker* functions' correctness.

Spawners

Methods in the *spawner* function group have an x and y-coordinate argument input, return a boolean and are public. GameObjects such as Traps of the Enemy object group and LifeEssences of the Reward object group are kept track of in their respective manager classes' linked lists, and *spawner* methods add new entries to those lists. Spawning initially checks the validity of the inputted coordinate and returns true if the object is successfully "spawned" or false otherwise. These methods reside in only two object groups: RewardManager of the Reward object group, and EnemyManager of the Enemy object group. There are two components to the testing of the *spawner* groups: the coordinate validity condition and the correctness of the actual spawn.

Regarding the coordinate validity, we can use unit tests and input known valid and invalid coordinates to verify the correctness of the *spawner*'s coordinate validity. Regarding the correctness of the actual GameObject, since we cannot automate for visual testing, we can assess the correctness via its interaction with other game elements, namely, the player. Hence, to test the correctness of the spawned GameObject, integration tests are needed.

Movers

Methods in the *mover* function group have no input argument and are private. These methods mainly revolve around wall collision detection and ensuring the corresponding GameObject objects do not exit the map. Since the methods are private, we can only test the correctness of the *mover* function through the update function and verify the coordinates or game state after each update. The *mover* functions reside in only two object groups: Hirsch, EvilSpirit and EnemyManager, of the Player and Enemy object group, respectively.

For the *mover* function in Hirsch, move()—its purpose is to prevent the player from walking through walls or through the entrances, via access of the isWall() and isEntrance() methods from the Map class, of the Map object group. We can verify its correctness by teleporting the player to a location on the map where it is immediately next to a known wall

or entrance and attempting to let the player walk through it. Due to the Map object group's involvement, this test is an integration test.

For the *mover* function in EnemyManager, mahanMove()—its purpose is to assign movement direction to EvilSpirit type GameObjects such that they can move closer towards the player. To test this method requires the additional participation of the PlayerManager class of the Player object group—making the test an integration test. We can verify the correctness of this method via examining if the player dies after a predetermined number of time steps.

For the series of *mover* functions in EvilSpirit, their purpose is to allow the EvilSpirit enemy type to occasionally have the ability to move through walls. When the ability is on, the EvilSpirits can move through any continuous wall, when it is off, they behave similarly to the player. Since this class is inherited from GameObject, we can verify its movement by getting its x and y-coordinate. However, since testing wall collision requires the access of methods from the Map class of the Map object group, any testing done here is also an integration test. Similar to the test for Hirsch, we can teleport the EvilSpirit object to a known location with an immediate wall on the side and have it attempt to move through it; we can wait out the skill cooldown via a sleep function. Additionally, we can test to ensure EvilSpirit does not move outside of the map.

Colliders

Methods in the *collider* function group have x and y-coordinate input arguments and are public. Object group with this function group includes EnemyManager, of the Enemy object group, and RewardManager, of the Reward function group. Together, they manage and initialize all GameObject to GameObject interactions in the game.

The EnemyManager consists of two *collider* functions: isEnemy() and isEvilSpirit(). Given an x and y-coordinate, isEnemy() returns the enemy ID of the GameObject, whose contact radius contains the given coordinate, null otherwise. This method can be simply unit tested for correctness by verifying that any coordinate within the contact radius of a Trap or EvilSpirit GameObject will return the correct ID. Given an x and y-coordinate, isEvilSpirit() returns the number of EvilSpirits whose contact radius contains the given coordinate; this method is utilized between the EvilSpirits to prevent overlap. Again, this can be simply unit tested for correctness by spawning multiple EvilSpirit and verifying if the method returns the correct value based on the inputted coordinates.

The RewardManager consists of one *collider* function: isReward(). Similar to isEnemy(), it returns the reward ID of the GameObject whose contact radius contains the given coordinate, then it immediately deletes that object from the game. Again, we can verify the correctness of this method via verifying that any coordinate within the contact radius of a

LifeEssence or ElusiveLifeEssence GameObject will return the correct ID and after the ID is returned, the object is deleted.

Resetters

Methods in the *resetter* function group have no input argument and are public. As the name suggests, they reset their respective classes by clearing existing objects and re-spawning any predefined objects. Examples of these methods include resetGame() in GameManager, and newGame() in all of RewardManager, EnemyManager, and PlayerManager. This function group will not be explicitly tested since it would be redundant as the *resetter* functions are used in many other tests cases. These methods are indirectly tested via other unit and integration tests.

Coverage and Quality

	Audio	Enemy	GameEngine	Map	Overlay	Player	Reward	Utility	Overall
Class Coverage %	100% (3/3)	100% (3/3)	60% (3/5)	100% (1/1)	0% (0/1)	100% (3/3)	100% (3/3)	100% (1/1)	85% (17/20)
Method Coverage %	90% (9/10)	86% (19/22)	73% (33/45)	90% (10/11)	0% (0/4)	93% (30/32)	86% (19/22)	100% (4/4)	82% (124/150)
Line Coverage %	79% (62/78)	89% (150/167)	46% (100/215)	83% (46/55)	0% (0/27)	77% (143/185)	90% (99/110)	100% (9/9)	71% (609/846)

As explained in the introduction, audio and visual elements are focused in our tests, and since each object group participates in in-game rendering, 100% code coverage is not always possible. Above is the report generated by Intellij; three object groups are worth noting here: Audio, Overlay, and Game. Regarding Audio, it is listed for a high percentage of convergence whereas in reality the Audio group was only indirectly “tested” via testings to the GameManager class which manages in-game sounds. There exists no automated testing verifying the “correctness” of the outputted audio, hence it should be zero. The Overlay group deals exclusively with on-screen elements, hence it was avoided. Two classes in the Game object groups were avoided due to their high testing complexity but triviality in manual testing: Game and GameWindow. GameWindow contains only one method in which it creates a JFrame window for our game to operate in, this can be easily verified through manual testing (i.e. if a window opens or not). The class, Game, contains the main function, frame incrementing logic, rendering logic, and initialization logic; each of these is difficult to test, but extremely easy to identify if there exists an issue since it acts as the GameEngine of our game.

With regards to quality, as noted in the introduction, we tried to make as few assumptions about the detail mechanisms of the game as possible, and coded our tests to be as flexible as possible. For example, currently, we have the damage cooldown period initialized on spawn (making the player invaluable during the first second), but in our test, we would examine both cases when the player is vulnerable on spawn and when it is not. For tests with large degrees of freedom like collision detection, we tried to create multiple tests, and at the very least examine the lower bound, upper bound, and a point in between.

Findings

While writing our test, we found that due to our code hierarchy structure, testing for the correctness of certain features can be difficult depending on the hierarchy level the feature is initialized in. For example, assessing if the KeyInputs class is correctly reflecting the user's corresponding actions (i.e. left arrow moves character to the left) was initially difficult as it was initialized on the PlayerManager level. To test the result, we needed to assess the player object itself (for information such as position and speed) which the PlayerManager only has private references to. One method was to forward certain information through new *getter* functions in PlayerManager, but we found that it was more efficient to move the initialization of KeyInput into Hirsch, the player object, itself.

Additionally, we realized there was a huge disconnection of information between the GameObject and the GameObject Manager classes (i.e. Traps to EnemyManager), making testing game states extremely difficult. Each GameObject Manager has private references to their respective GameObjects with little ways for external access. To help us evaluate or make possible certain integration tests, we introduced public *spawner* methods that allow for external control of the spawning of GameObjects; this made testing victory or defeat states (which require information of various GameObjects onboard) far simpler. Furthermore, we were able to add another layer of protection for invalid inputs, as now every spawning attempt (including internal calls) would go through the *spawner* methods which contain various error checking and assertions.

We were also able to find an issue in our testing with regards to the Map class. We noted that it had no self-protection for out-of-bound coordinate inputs in isWall() and isEntrance(). This was a vulnerability in the game that could cause it to crash. For example, if an EvilSpirit object activates its wall-walk skill and was moving along the border of the map, this would cause an out of bound error due to our 1:20 map matrix compression. We were able to quickly fix this issue through adding new error checking if-statements.