Phase 3 Report

Group 25

Vincente Buenaventura
Noah Kamzelski
Samin Moradkhan
Lionel Song

Tests

Unit Tests

PlayerTest class

- Assert that player moves up, down, right, left by set movement speed
- Assert that player cannot move when colliding with collidable tile/object
- Assert that player reset function resets to correct position on the map
- Assert that colliding with banana increases score by 20
- Assert that colliding with apple increases lives by 1
- Assert that colliding with trap decreases lives by 1
- Assert that colliding with an unknown object does not change score or lives
- Assert that colliding with enemy decreases lives by 1, and resets player position
- Assert that colliding with a disabled enemy does not change lives or reset player position
- Assert that the function which checks whether the player has lives returns the correct bool
- Assert that player hitbox is correctly initialised
- Assert that player hitbox can change size correctly

EnemyTest class

- Assert that enemy is randomly moving in a set direction
- Assert that enemy colliding with player decreases player lives by 1
- Assert that enemy colliding with player resets their position
- Assert that enemy hitbox is correctly initialised
- Assert that enemy moves up, down, right, left by set movement speed
- Assert that enemy changes direction every second
- Assert that enemy cannot move when colliding with collidable tile

EntityListTest

- Assert that an object can be successfully added to the object arraylist
- Assert that the “object clear” function clears all objects in the arraylist
- Assert that the “object list size” function returns the correct arraylist size
- Assert that the correct object can be retrieved at specific index
- Assert that the correct object can be deleted at specific index

- Assert that an enemy can be successfully added to the enemy arraylist
- Assert that the “enemy clear” function clears all enemies in the arraylist
- Assert that the “enemy list size” function returns the correct arraylist size
- Assert that the correct enemy can be retrieved at specific index
- Assert that the correct enemy can be deleted at specific index

AssetCreatorTest

- Assert that a specific coordinate already has an object in that location
- Assert that a tile exists so the object can be placed
- Assert that a set of objects get created
- Assert that a set of enemies get created
- Assert that a set of apples get created
- Asset that a door gets created

KeyboardTest

- Assert that the keys for moving upwards (W or up arrow) have the same key code as the keyevent button pressed
- Assert that the keys for moving downwards (S or down arrow) have the same key code as the keyevent button pressed
- Assert that the keys for moving left (A or left arrow) have the same key code as the keyevent button pressed
- Assert that the keys for moving right (D or right arrow) have the same key code as the keyevent button pressed

SimulatorTest

- Assert that the game gets setup
- Assert that the game stats get reset by checking the lives of the player is back to full
- Assert that the games’ objects get reset by confirming the coordinate of the players’ location
- Assert that the screen width is correct
- Assert that the screen height is correct
- Assert that the size of the tiles are 48x48
- Assert that the player object isn’t null or empty
- Assert that the default player location X position is the starting cell
- Assert that the default player location Y position is the starting cell
- Assert that getting the list of entities retrieves the correct information
- Assert that starting the game thread is the same as making a new game thread
- Assert that the game gets updated with the correct values by checking the size of the object list
- Assert that playing the sound effect chooses the correct one corresponding to the input
- Assert that the players position has reset by checking with the starting cell
- Assert that the players’ lives have been added by the inputted integer

SoundTest

- Assert that the sound effect played when the player collects the banana is the same as the sound effect stored in the array of sounds
- Assert that the sound effect played when the player collects the apple is the same as the sound effect stored in the array of sounds
- Assert that the sound effect played when the player is hit by the enemy or stuck in a trap is the same as the sound effect stored in the array of sounds
- Assert that the sound effect played when the player reaches the end of the game (door) is the same as the sound effect stored in the array of sounds

Integration Tests

Simulator and other classes

- Testing to reset the players position as well as the score and life of the player when the simulator restarts (as the player loses the game)
- Testing the sound effect of the player when its moving in the game without colliding with any other object

Player Class and other classes

- We have the player and the keyboard test integrated together so when the keyboard is pressed the player moves in the desired direction.
- The player and the check collision class which is called whenever the player hits the rewards, enemy, trap or the final end mark. Also when the player hits the wall surrounding the map because it cannot go further from the walls.
- We also test when the player does not hit any other objects which means it should continue its path without playing any sound effect or being repositioned because of hitting an enemy or a trap.
- The sound class is also tested with the player and the collision because that is how we show that the player is hit by another object.

AssetCreator and other classes

- The asset creator test class tests that the objects like apple, banana, enemies and door are actually created.
- This is an integration test between the asset creator class and the object package as well as the entity list which has the subclass of animate entity and is inherited by the enemy class.

Enemy class and other classes

- In the enemy class we also test the position of the enemy and the moving direction which are all implemented in the coordinates class.
- We also test the collision here because if the enemy collides with the wall it cannot go through or move further.

The majority of the code in our CheckCollision class needed various components such as access to the Player, Simulator, and EntityList class. Many of our tests that are written in the Player and Enemy collision tests are passed both the Simulator and CheckCollision class. The CheckCollision class would then get the entity's hitbox and check if the hitbox intersects with each other, then return a bool.

In general most of our test cases have other classes integrated with them because most of the features are dependent on each other and therefore a thorough test would be a test that can check the functionalities all together. We tried our best to make distinct test cases where we assess all the different possibilities for the outcome.

Test Quality and Coverage

In terms of the code coverage, we have made use of some plugins that are offered by IntellijIDEA. Using this feature we are able to find the code coverage percentage for each class separately as well as the classes tested while testing a separate class (integrated testing). From the results, we see that all of the methods in most of our classes except for the player class are tested therefore a 100% code coverage is shown for all of them. The only method that was really difficult (or impossible) to test is the draw function and that is due to not being able to call the draw function without having all the game setup since this function is responsible for uploading the picture of the moving characters as they change direction.

One of the other classes that is difficult to test is the UI class. This class is dependent on the rest of the game, which means we have to have all other components working together to be able to test the visuals. It is not easy to test our user interface without having the simulator running and loading the components of the game. Since that is the case, this class does get tested indirectly with the other classes but we cannot test specific functionality of it. Our simulator class has a method that calls the majority of the methods that are in the UI class. Even though we weren't able to make test cases for the UI class, it does get tested using the other classes that it depends on.

In terms of the branch coverage, we tried to have test cases where we can test any possible outcome of the game. For example we test if the objects actually collide when the main player collects the rewards, is hit by enemy or trap or reaches the end mark. Upon collision we play the sound effect to make sure that happened. We also make sure to add to the player's lives when it collects the bonus reward and add to its score when it collects the regular reward. If the objects do not collide, the score and life should remain the same and also there is no sound effect played. The rest of the tests also follow the same pattern. As another example for the keyboardTest we compare if the pressed key has the same keycode as expected but we also compare that another key that is not used in our game should not have the same keycode as the other ones. In this way we test the branch coverage. Of

course it is not accurate to give an exact percentage for the branch coverage but from what we calculated by looking at our test cases, 60% coverage is evident.

Findings

What you learned from writing tests? Any changes to the original code?
One thing we learned is that messy code makes creating unit and integration tests much more difficult. Oftentimes, old and clunky functions with multiple parameters had to be split up into smaller functions to make testing viable. Large switch statements had to be split and optimised to prevent clustered and redundant tests.

Some important bugs we found and fixed:

- Unresponsive end goal/goal tile

Sometimes when the player reaches the goal tile, the player would not be able to proceed to the next level. Our solution was turning the goal tile into a collidable object door that triggers the next level screen. Since unit tests have already been done for player to object collisions, it was easy to verify that the door was working.

- Apple random spawning malfunctioning.

By our use case, apples should be able to spawn randomly as well as appearing and disappearing throughout the game. Sometimes, apples would stop appearing/disappearing.

- Traps blocking pathway/corridor, leading to block route.

Sometimes the randomly generated traps were placed in a way that blocked the player or prevented the player from winning. We now implemented a function that prevents traps from spawning nearby each other.

- Traps spawning on player start position and end goal

While implementing our unit tests for entity spawning, we found that the traps had a chance at spawning on the player's start position or end goal, making the game impossible to win.

- Player doesn’t lose when lives go to zero

Our player unit tests found that the player would not lose when reaching zero lives, and could have negative lives. Now, players lose the game as soon as their lives reach zero.

- Improvements to our UI design

Our UI design was very simple at first, we made some enhancements to make it look more professional and better to present.

- End game phase reset

Now when we reach the end of the game everything in the game resets including player score, health, time and the map resets to the first map.