Phase 3 Report -- Group 13

Features that needed to be unit tested and briefly explain these features:

Individual classes and their methods that could easily be checked for expected values and behaviour through the use of asserts were tested using unit testing. Constructors were tested by using assert on the values that were expected to be initialized. We went through each class and unit tested each method, when possible, because each method would eventually be used by another class. The methods tested were the ones that didn't have interaction with objects of different classes and we can manipulate the values to check that the method will behave in an expected manner.

Important interactions between different components of your system:

- Player running into…:
    - Regular reward:
        - Player wins 5 points for collecting each regular rewards
        - Player needs to collect all 10 regular rewards to win the game, which is implemented in Gatherer class.
        - Regular rewards disappear after being collected which is implemented in the Gatherer class.
    - Bonus rewards
        - This is an extra challenge for the player. It's an extra way to gain more points and beat your personal high score.
        - Bonus rewards disappear after being collected, which is implemented in Gatherer class.
        - Bonus rewards disappear after 20 seconds, which is implementing in GameWindow class
    - Scavenger
        - Player lose the game once Scavenger collides with Player, partially implemented in CollisionChecker class.
    - Trap
        - Player lose 10 points once they collide with each trap.
        - Traps disappear after being collected, which is implemented in Gatherer class.
    - Wall
        - Player and can not walk through walls, implemented in CollisionChecker class.

- Scavenger running into the walls
    - Scavengers cannot walk through walls implemented in CollisionChecker class.

Test case/class covering which feature/interaction:

--which test case/class covers which feature/interaction

- Unit tests:

    - The unit tests for each class were named after the class that they are testing with the added ‘Test’ to the end of the name
    - For each method tested, it was also named after the method being tested with the added ‘Test’ at the end

--Measure, report, and discuss line and branch coverage of your tests.

- Scavenger:
    - Line coverage: $\frac{28}{105} = 26\%$
    - Branch coverage: $\frac{1}{34}$
    - ❖ Poor coverage overall because the main lines of code in this class are in the functions that couldn’t be tested with unit testing
- Trap:
    - Line coverage: $\frac{4}{5} = 80\%$
    - No branches in this class
    - ❖ Since this class only consisted of the constructor, most lines were able to be asserted. The one line that we could not assert was the line where an object from java.awt was declared and initialized within the constructor and could not be accessed outside.
- EndTile:
    - Line coverage: $\frac{4}{5} = 80\%$
    - No branches in this class
    - ❖ Since this class only consisted of the constructor, most lines were able to be asserted. The one line that we could not assert was the line where an object from java.awt was declared and initialized within the constructor and could not be accessed outside.
- RegularResource:
    - Line coverage: $\frac{4}{5} = 80\%$
    - No branches in this class
    - ❖ Since this class only consisted of the constructor, most lines were able to be asserted. The one line that we could not assert was the line where an object from java.awt was declared and initialized within the constructor and could not be accessed outside.
- BonusResource
    - Line coverage: $\frac{4}{5} = 80\%$
    - No branches in this class
    - ❖ The one line that we could not assert was the line where an object from java.awt was declared and initialized within the constructor and could not be accessed outside.
- Game_Tiles
    - Line coverage: $1/2 = 50\%$
    - No branches in this class
    - ❖ The one line that we could not assert was the image attribute because it is set to null at first and later specify in Tile Manager

- Tile_Manager
    - Line coverage: 9/42= 21%
    - No branches in this class
    - ❖ Poor coverage overall because we leave the drawing images method to be tested manually.
- CollisionChecker
    - Line coverage: $272/331 = 82\%$
    - No branches to be tested
    - ❖ The collisions between gatherer and rewards and between scavenger and gatherer is checked.
- Input
    - Line coverage: $9/14 = 64\%$
    - Branch coverage: 100%
    - ❖ The boolean values are checked for when the key is pressed and is released.
- AssetSetter
    - Line coverage: $40/61 = 66\%$
    - No branches in this class
    - ❖ Checked whether the resources and scavengers were set on the maze on the window.
- Object
    - Line coverage: $1/3 = 33\%$
    - No branches in this class
    - ❖ Tests the member variables for their default values and verifies that they are set correctly.
- GameWindow
    - Line coverage: 22/51 = 43.1%
    - No branches tested
    - ❖ Tests to make sure the graphics are displayed properly and in the right resolution/size.
- Gatherer
    - Line coverage: 21/50 = 42%
    - No branches tested
    - ❖ Ensures that the player character spawns and interacts properly with the environment and responds to player input as designed.
- Entity
    - Line coverage: 10/34 = 29.4%
    - No branches tested
    - ❖ Tests to make sure all the entities spawn appropriately and exhibit the properties that were given to them.
- UI
    - Line coverage: 4/15 = 26.7%
    - No branches tested

- ❖ Ensures that the user interface displays all the necessary components such as notifications, timer, and score.

Measures for ensuring quality:

- We create unit tests for all classes (except the main class)
- Try to test all possible values (not just the expected values but weird ones that are highly unlikely to appear)
- We go over all the requirements mentioned in Phase 1 and Phase 2 and make sure the game has all the required functionalities.

Features or code segments that are not covered and why:

- We weren't sure how to write out integration tests and so we concluded that we would manually conduct these tests. We think that by manually testing it would be more true to how a player may encounter possible issues.
- The Main class:
    - This class was mainly setting up the GUI and the game itself by calling methods of other classes.
    - No tests were written for this class because it can be visually seen if it worked or not.
- In Scavenger:
    - draw()
    - update()
    - We test these two methods manually by executing and playing the game because it is easier to know whether it's working by seeing it in action.
    - The if/else statements in setAction(): this part relies on a random value that cannot be manipulated externally
- In Tile Manager:
    - getBGImg() and draw()
    - These two methods aim to get image resources and draw images. We manually test these two methods by executing and playing the game.
- In Game Tile:
    - We leave the Image attribute because it is set to null at first and later specify in Tile Manager
- In Object:
    - draw()
    - We test this method manually by executing and playing the game because it is easier to know whether it's working by seeing it in action.
- In CollisionChecker:
    - Local variable
    - The values are dynamic and change with every 'tick' of the game.
- The GameStat class and GameState class:

    - We leave these two classes because it is entirely graphic and update methods. We test these two classes manually by playing the game because it is easier to know whether it's working by seeing it in action.
- In Entity:
    - Sprite images
    - These are displayed when running the game and therefore tested manually
- In Gatherer:
    - update(), draw(), pickUpObject(), and interactEnemy()
    - The values in these functions are dynamic and are updated with every tick as the game runs
- In UI:
    - draw() and draw_menu_screen()
    - Since the values are dynamic, the only way to test the functions would be to run the game itself
- In GameWindow:
    - run(), paintComponent(), and update()
    - The values change during runtime of the game

Important findings:

Writing and running tests is a great way to go over the code once again because when going in with the mindset of writing tests, you kind of see the code differently. This allowed us to notice the redundant and unnecessary parts of our codes and thus we deleted all the redundant methods when revising and testing the code. We fixed any bugs when they appeared in both unit tests and manual tests, so our game implements the correct functionalities.