Project Phase 3 Report Group 20: Escape From Corona

Adjustments to code to allow for greater testing

This phase focused on writing test cases, though before getting started on testing, we had to ensure that our program was at a level that we were satisfied with. One major issue that we had noticed before and while writing tests was the lack of modularity. We addressed this problem by breaking up some of our methods into smaller components, and even dividing our God class into smaller classes to allow for more balanced testing and lower coupling. Many classes within the program had to be fitted with setters and getters to allow us to test variables throughout. By pursuing these steps before and while testing, it allowed for greater code test coverage and better overall design as well.

Additionally some aspects of our code were clearly identified as bad code smells, and we took necessary steps to address and resolve the problems. For example, some code that fell within the realm of Feature Envy in our Board class had to be refactored by Move Method to the Game class, to allow for better interactions and function calls. There was also a lot of code in the Game class that was refactored into helper functions, and cleaned up so as to reduce the feeling of a "blob."

Improvements to actual game and redesigning UI; Findings

In addition to major structural overhaul, we decided to revamp and improve our UI and menus, adding a theme and clear aesthetic to our game to make it more user-friendly. Start-game, end-game, and pause-screens were fitted with their respective assets to provide a more cohesive and comprehensive game experience to the player. Additionally, features such as dynamic resizing of the game and centering of the game within the window were added to again provide a cleaner interface.

One major feature that was implemented was the addition of extra levels, upon which the player would have to complete all levels before actually completing the game. This allowed to a progressive scaling of difficulty throughout, and overall provided a better experience. While developing this portion, we made sure that our code stayed modular, and as a result of that, we found implementations of new levels to be extremely easy, typically just adding in a new 2D array representing the level design, and adding methods to load that array.

Additionally, sound effects and audio were also added to provide a more immersive experience, this was done through using Clip, AudioInputStream, and AudioSystem in Java. Button clicks, button hovers, startGame, and endGame all felt more responsive as well as a common game "theme song" being played throughout gameplay.

We made a conscious effort to stray from the given requirements stated in phase 1 of the project, and adjusted tick speed such that the main character would move faster than enemies. This allowed the player to have a more fun experience and actually allowed the player to win the game.

While we were implementing all these features, we identified many bugs that were causing our program to lag, such as loading a FileInputStream for every tile, every tick instead of just passing a pre-loaded asset instead. The same was done for all sounds using AudioInputStream. Upon resolving this issue and others, our game ran noticeably faster and with improved responsiveness; by thoroughly observing the code, we also managed to squash some persistent bugs. Implementing tests significantly improved the quality of our code by pushing towards a modular design and consequently improving readability.

Features that needed to be unit tested

All classes within the src/main folder needed to be tested, though depending on how they were structured, they were tested differently. As an example, classes in the TileAction package were easy to unit test as they were extremely modular, though a class like Game proved to be extremely difficult to identify tests as it handled all items pertaining to JavaFX. The following features were the ones that needed to be unit tested:

- In the BoardDesign package, we needed to test the open state of the Entrance, Exit, and Wall, as well test that each one upon initialization wouldn't have a reward. Additionally, Exit's state was checked when the checkpoints left variable was adjusted. We attained essentially a 100% line coverage in this package.
- In the Characters package, it was more complicated as we needed to check integration tests that would check both Enemies and the MainCharacter's positions in different movement scenarios. Enemies were checked using their checkBestMovement and get position methods, with a Mockito created Board, and integration tested on the Board with wall tiles next to them to see whether their movements changed correctly. The MainCharacter was tested similarly, though instead of using a checkBestMovement method, we instead simulated Key press events using a KeyEvent object, using corresponding keys. Additionally, since MainCharacter is a singleton, we had to check whether the functionality of MainCharacter was working as intended, and in line with the design pattern description.
- In our Core package, we encountered the greatest amount of issues. Our Board and Tile classes, while complicated, were able to eventually be unit tested when modularized enough, and we tested all instances of different board generation and tile generation. The main problems came from our main UI class, which was our Game class. While we were able to identify some features that could be unit tested, a lot of it required other components. This is where many integration tests were made to check behaviour when

other objects and classes interacted with the Game class's methods. We tested enemy generation, as well as primitive values after methods were executed to check whether things had worked out correctly. Additionally, we simulated the escape key press, similar to how we tested in our MainCharacter class.

- Our TileAction package was relatively smooth sailing. All classes within this package were tested to check whether their score increase worked, and check whether their generated x and y coordinate were correct. Additionally, Bonus's tick decrement was checked, and Checkpoint's checkpoint count was checked as well when a checkpoint was stepped on.

Important interactions between different components of the system

- Many integration tests were implemented to see interactions between classes from different packages.
    - We tested:
        - Main character interaction with checkpoints
        - Main character interaction with bonus
        - Simulated an entire game to check whether the player can win the game and if exit opens upon checkpoint collection by main character
        - Simulated a lose game event when the main character encounters an enemy.
    - Additionally, since some tests within each test class utilized multiple classes to find a test result, those could also be considered integration tests.
- The Game class, as the main UI class that interacts with many components to draw on the JavaFX window, contains many integration tests that check whether those interactions do not have any errors and provide a sound result.

Test Automation

All tests can be run with Maven, with tests placed in src/test/java, though we had no test resources. The command that is used to run the tests is "mvn test", and it runs through all our unit tests and integration tests.

Test Quality and Coverage

According to IntelliJ, which has a built-in feature to test line coverage, we had greater than 90% test coverage excluding the GUI. However, we ran into issues testing things related to the application platform (JavaFX) and UI. Some stuff, like testing whether objects were placed on the screen, or if the text displayed correctly, or checking whether the enemies were properly displayed, were not possible to be tested using the testing frameworks that we used. Instead, we employed manual testing to check that everything had executed correctly regarding the UI, screens, buttons, and sound effects. The graphical nature of the UI made 100% coverage impossible.

Features that have been unit tested

- BoardDesign
    - Entrance
        - Test open of the entrance
        - Testing that the entrance doesn't have a reward
    - Exit
        - Test open if conditions met
        - Test not open if conditions not met
        - Testing that exit doesn't have a reward
    - Wall
        - Test if it is open (it shouldn't be)
        - Testing that wall doesn't have a reward
- Characters
    - Direction
        - Check if getVal method returns correct enum values
    - Enemy
        - Test if move method updates object coordinates appropriately for valid and invalid inputs
        - Check if checkBestMovement method returns the appropriate Direction with different scenarios according to the algorithm
        - Check if checkBestMovement method returns the appropriate Direction when blocked by another Enemy object
        - Check if isColliding method returns appropriate bool
    - MainCharacters
        - Check if getMainCharacter returns the same object
        - Check if restartMainCharacter returns a MainCharacter object with appropriate coordinates specified
        - Check if isColliding method returns appropriate bool
        - Check if keyPressed method updates object positions appropriately with valid and invalid inputs
        - Check if keyPressed method keeps current positions if input would make coordinates go out of bounds
        - Check if keyPressed method keeps current positions if input would make object collide with an unopened Tile
    - NonStationaryCharacter
        - Check that positions are set correctly
- Core
    - Board
        - Test that no tiles are null when instantiated

        - Test bonus generation
        - Test switch statements by passing in a 2d array that is 1x1, check that the resulting board is a 1x1 board of the type of tile that we wanted** NEXT
        - Test the exitXPos when passing in a board with an exit
        - Test exitYPos when passing in board with exit
    - AssetLoad
        - Test if assets are null if path not valid
        - Test if assets are not null if path valid
    - Game
        - Test enemy generation on each stage
        - Test startGame
        - Test endGame
        - Test restartGame
        - Test updateScore
        - Test escape key pauses game
        - Test pressing escape key unpauses game
        - Test that application launches

    - Tile
        - Removing the Reward
        - Testing False Input for Switchcases (none of the first three)
        - Checking whether Reward type is Bonus
        - Checking whether Reward type is Checkpoint
        - Checking whether Reward type is Punishment
- TileAction
    - Bonus
        - Test that it increases score
        - Test tick decrement
        - Test x and y coordinates
        - Test that the update playerscore is actually updating the game score
    - Checkpoint
        - Test that it increases score
        - Test that on creation of new checkpoint, global checkpoint count goes up
        - Test decrement of checkpoints
        - Test that checkpoint count is actually being decremented on update score
        - Test that the checkpoint increase in score is correct
    - Punishment
        - Test that it decreases score
        - Test that update player score is actually updating the game score

Integration Tested

- MainCharacter and Bonus → score
    - Check score updated appropriately when player interacts with tile
- MainCharacter and Checkpoint → score
    - Check score updated appropriately when player interacts with tile
- MainCharacter and Punishment → score
    - Check score updated appropriately when player interacts with tile
- MainCharacter and Exit Win
    - Check if exit opens when player collects all checkpoints
    - Check if game ends when stepping on exit
    - Check if timer is stopped
- MainCharacter and Enemy collide
    - Check that the game ends and player loses when enemy collides with player