Assignment 4: Report

Homework 4: Code review report

Annu Semwal (301566506)
Nazanin Pouria Mehr (301442860)

In our code review for the project, we found several issues that made the code harder to read, understand, and maintain. The Game class, in particular, was doing too much and had become too large (God class). To fix this, we split its responsibilities into smaller, focused components. For example, managing background music was originally handled by a method within the Game class. This made the class unnecessarily complex. To solve this, we created a new Sound class that specifically handles all sound-related tasks like the background sound. This change helped make the Game class smaller and easier to understand while also improving modularity by keeping sound-related logic separate.

The constructor in the Game class was also a problem because it was too long and tried to do too many things at once. A constructor should mainly focus on initializing the class, but ours included extra logic, like setting the background and creating enemies. To address this, we refactored the constructor by moving the background setup into a separate setBackground method. Similarly, we created a createEnemies method that handles all the logic for generating enemies. These changes made the constructor much shorter and clearer, so it's now easier to see its primary purpose: setting up the game.

We also noticed that some methods in the Game class were overly long and combined multiple tasks into a single method, which made them hard to follow. For example, the getTileImage method was doing too much at once. To fix this, we broke it down into smaller, focused methods, each handling one part of the process. Another example was the readImage method, which lacked proper error handling. We added exception handling to make it more reliable, so the game won't crash if there's an issue with loading an image. Additionally, we introduced a new scaleImage method to handle image scaling. This kept the scaling logic separate from other methods, making the code easier to follow.

In the Menu class, we had a method called ActionPerformed(ActionEvent e) that was far too long containing too may if/ if else/ else statements and tried to handle so many responsibilities in one place. This made it confusing and hard to modify. To improve it, we broke this method into smaller sections, with each section focused on a specific part of the menu's behavior. Now, the Menu class is more organized, and it's much easier to understand what each part of the code is doing.

Noticed Data Clumps throughout the code, such as Menu classes and Game class. To deal with data clumps, we used Extract Class method to extract result variables into a Class GameResult and created an Instance of the class in the Menu classes.

The constructors of Menu and EndMenu classes give of a few code smells, such as Long Method and also violates the “Single Responsibility Principle”. To fix these code smells, we use the Extract Method Pattern to extract the functionality of initializing Buttons and Labels into their own methods. We created methods like createButton and createLabel in both Menu and EndMenu classes to ensure the persistence of the single responsibility principle. We noticed this method could also be extracted into a separate class, but since we only have 2 instances of copied code, the rule of 3 does not invoke.

In the Menu and EndMenu classes, we fixed code smells like Duplicated Code, Data Clumps, Long Methods, feature envy, etc. with the use of methods such as Extract Class and Extract Method. We were able to make the functionality of both classes much better and much more open to changes and less likely to break the code. Refactorization was done without the loss of any original behaviour which made it a success.

In our enemy class we have refactored the enemy movement code to make it easier to understand and manage. Instead of having all the logic in one big block, we broke it into smaller methods. For example, we created `isOutOfRange()` and `isWithinAggroRange()` to check if the enemy is too far or close to the player, which makes the main update method simpler. we also moved the movement code into separate methods: moveInFacingDirection() moves the enemy based on which direction it's facing, and `moveTowardsPlayer()` moves the enemy toward the player, deciding which direction to move based on the player’s position. This makes the code cleaner, easier to read, and easier to fix or update in the future, without changing how the enemy moves.

Overall, our refactoring focused on solving specific problems, such as long methods, large classes, code duplication, data clumps, feature envy, low cohesion and many other minor code smells. By splitting up responsibilities and moving related tasks into their own classes or methods, we improved the design of the code. These changes not only made the code easier to read and maintain but also made it more robust and prepared for future updates. Every change was tested after implementation to ensure the program's behavior stayed the same, and we committed our work regularly to keep track of progress.