Assignment 4 Report
CMPT 276 - Fall 2023

Group 24

Raymond Chan
Timmy Tsai
Due: December 5, 2023
School of Computing Science
Simon Fraser University

Unnecessary switch/case statements (Branch: newRefactoring, **Commit:** cf9c50d**)**

- The pickUpReward method located in Main_Character.java had too many case statements for each object that the main character could pick up. To address this issue, a map to store reward actions associated with their names called rewardActions was created along with a RewardAction interface to enforce a common structure for each reward action. Each action now implements the RewardAction interface. For example, RegularRewardAction, BonusRewardAction, PunishmentAction, etc. This approach encapsulates the specific logic for each reward type within its corresponding class. A method called initializeRewardActions was created for initializing the rewardActions map with instances of the concrete RewardAction classes. The constructor of Main_Character.java is then modified to include the initialization of rewardActions using the initializeRewardActions method.

Duplicated Code (Branch: newRefactoring, **Commit:** 985d2c6**)**

- The setReward method located in RewardSetter.java contained many instances of duplicated code to set the type of reward and its location. The refactoring of the original code involves replacing hard-coded assignments for each game object with a more data-driven approach using an array. In the refactored version, a 2D array rewardData is introduced to store information about each reward, including its type, column, and row. This allows the code for easier maintenance and future updates. The loop in the refactored version iterates over the rewardData array, setting the rewards dynamically based on the provided data. The refactored code is more readable, especially when dealing with a larger number of rewards or frequent changes to the reward positions. The same refactoring was also done to PunishmentSetter.

Long Method (Branch: newRefactoring, **Commit:** 988bcaf**)**

- The update method in Main_Character.java was quite long and contained a mix of responsibilities. In the refactoring process, the update method of the MainCharacter class was systematically broken down into smaller, more focused methods to enhance clarity and maintainability. The original method, which handled various responsibilities such as movement, collision checking, sprite updates, and oxygen management, was divided into different methods. The updateDirection method was introduced to handle the logic of updating the character's direction based on keyboard input, while the handleObjectCollision method deals specifically with object interactions. The moveCharacter method focuses on character movement, and updateSpriteCounter and toggleSpriteNum methods manage sprite animations. Additionally, the updateOxygenTimer and decreaseOxygenLevel methods were created to encapsulate the oxygen-related functionality.

Poorly Structured Code (Branch: newRefactoring, **Commit:** 7500f89)

- The SuperObject class's draw method is now more straightforward. The code was improved by separating the part responsible for adding a blue tint to the image into a new method called applyBlueTint. This adjustment shortens the draw method and keeps its focus clear.

Break down Method in Keyhandler.java and Ui.java (Branch: main, Commit: c53792d)

- In "KeyHandler.java" and "Ui.java," there was a long method called "KeyPressed" that did many things. To improve it, I broke it into smaller methods, each handling specific parts of the game or a key press. This makes the code work better and lowers the chances of mistakes.

- For instance, for the "KeyPressed" method, I decided to create separate methods: "handleTitleState" for the title screen and "handleEnterKeyInTitleState" for the Enter key just for the title state. This makes the code easier to read and understand instead of a big chunk of instructions inside “KeyPressed.”

- In the "Ui" class, I saw the "draw" method had a lot of drawing instructions. To simplify, I split it into smaller methods. For instance, I made "drawPlayState," similar to "drawPauseScreen" and "drawWinScreen." This organizes the code better, showing which part of the game is drawn at each step. Breaking big methods into smaller ones not only helps readability, it also helps developers easier to detect bugs and even continue to work on the project.

Improve variable names, methods names, and comments: (Branch: main, Commit: c53792d)

- It's important to use names for things in computer programs that are easy to understand. This includes the names of variables and methods. In the "Ui" and "KeyHandler" parts of the program, I looked at these names and made them clearer so that anyone reading the code can quickly figure out what they're for. I also added comments to explain more about what each part does. Using good names and helpful comments makes it easier for different people working on the program to understand and work together. It helps avoid confusion and mistakes, making the whole process smoother.

Improve if statement and for loops: (Branch: main, Commit: c53792d)

- The existing code contained multiple if statements that sequentially checked through each condition, potentially impacting efficiency. To enhance readability and optimize the code, I made an array named texts to store the strings to be displayed on the UI. Essentially this array of text is looped through using a For loop, which reduces the repetition of the code but also makes it more maintainable. This approach improves efficiency by reducing the need for multiple individual if statements and enhances the scalability of the code, as adding or modifying instructions in the future becomes easier and less error-prone.

Remove unused code: (Branch: main, Commit: c53792d)

- Duplicated and unused code, where similar or identical code segments appear in multiple places, can lead to maintenance challenges and increase the chances of bugs when changes are made. I identified and refactored duplicated code segments, consolidating them into reusable methods or eliminating redundancy. This not only improved code maintainability but also reduced the chances of introducing inconsistencies or errors during future modifications. This refactoring ensures that updates or fixes only need to be applied in one place, promoting code coherence and minimizing the risk of introducing discrepancies.