Yanjun and Jonas
CMPT 276 Assignment 4
April 11, 2024

Yanjun:

-Large Class: “GamePanel” class contained an excessive number of variables and methods, making it a "God Object" — a class that does too much.

- -Refactor Step:
    - -Extract Interface: Created an interface named ScreenSetting that encapsulates all final variables related to game screen settings such as tileSize, screenWidth, and screenHeight.
    - -Extract Abstract Class: Developed an abstract class to house common properties and methods for setting up a game panel. This includes initialization, getters for hero, enemy, and items, and foundational game methods like startGame and endGame.
    - -Inheritance Hierarchy Refactoring: Refactored MainGamePanel to inherit from the new abstract game panel class, promoting code reuse and organizational clarity.
- -Improvement:
    - -The GamePanel class now has a more focused role, with screen settings abstracted to an interface and shared functionality moved to an abstract class. This makes the system more modular and easier to manage.

Duplicate Code: The “setImage” method was duplicated across multiple classes that handle characters and items. This duplication leads to increased maintenance effort and potential for inconsistencies.

- -Refactor Step:
    - -Extract Method: Created a new method in a class named UtilityTool to handle the loading and setting of images from resources.
    - -Utilize Inheritance/Composition: Refactored character and item classes to use an instance of UtilityTool for image setting tasks, thereby removing the duplicate setImage methods from these classes.
- -Improvement:
    - -The refactoring has eliminated the duplicate setImage methods across various classes, centralizing the image loading logic within a single UtilityTool class. By using the UtilityTool class, we promote code reuse, making the overall codebase cleaner, more maintainable, and easier to understand.

Long Method: The “update” method was too lengthy, making it hard to understand and maintain. It included several responsibilities, like handling collisions, which could be abstracted out.

- -Refactor Step:
    - -Extract Method: Split the collision-related responsibilities into separate methods - “checkTileCollisionAndMoveHero”, “checkEnemyCollision”, and “checkItemCollision”.

-Create Wrapper Method: Developed a new method “collisionCheck” to encapsulate the three new methods related to collision checking.

-Replace Code with Method Call: Substituted the collision-related code in the update method with a single call to the new collisionCheck method.

-Improvement:

-The update method's complexity is significantly reduced by abstracting out the collision detection and handling into separate methods. This makes the update method shorter and more readable, adhering to the Single Responsibility Principle. It also makes the code easier to test, as collision logic is now isolated and can be tested independently.

-Duplicate code/switch statement: In the “draw” method within the abstract Character class. The assignment of currentImage occurs multiple times and the switch statement overuses for a simple condition.

-Refactor Step:

-Extract method: Isolated the logic for setting currentImage into a new method updateCurrentImage.

-Replace Conditional with Polymorphism: Use a ternary operation instead of switch for simplicity.

-Improvement:

The draw method now solely focuses on drawing, and the logic for determining the image has been abstracted to another method. Also, eliminating the switch statement simplifies the control flow, making the code less prone to errors during future modifications.

Jonas: