Code Review - Group 6

Natalie, Rebekah

During the code review process, various code smells and other features were found and changed accordingly, as detailed below. A separate branch was created on GitLab called “refactoring-NR” for code reviewing and refactoring purposes, and any changes made were correspondingly committed and pushed to the master branch to further improve the structure of our code.

Unused Variables

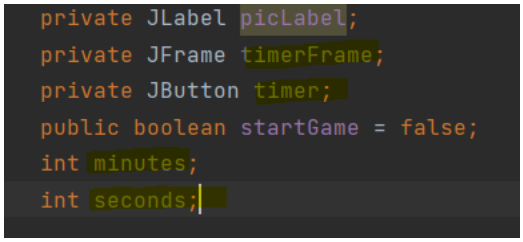

Figure 1: Code snippet containing unused variables in the Game class.

As highlighted in Figure 1, there were unused variables left in the Game class from an attempted implementation of the in-game timer. The original justification for implementing the timer in the Game class was that the Game class contained a startTime variable that kept track of the time elapsed since the game started, which was necessary to compute the time. However, the timer implementation was later changed to the GameScreen class because it handled all of the updates for each tick of the game, which made more sense as the timer drawn on the screen had to be refreshed. This implementation involved getting the startTime variable from the Game class instead. Nonetheless, the unused variables from the initial implementation, alongside commented out bits of code leftover after switching to the implementation in the GameScreen class, were removed to improve the clarity of the written code. Additionally, the startTime variable was actually named “starttime” prior to the code review and was accordingly changed in order to match the camel casing of the rest of the code to be consistent in formatting.

Dead Code

As the development of the project progressed, some methods became unused due to alternative implementations of the game and were hence removed by refactoring. For example, the implementation for enemy movement was changed from tile-by-tile movements to a more continuous pathfinding algorithm and was thus given an entirely new set of code—hence rendering the movement code previously found in the Enemy class irrelevant.

Duplicated Code

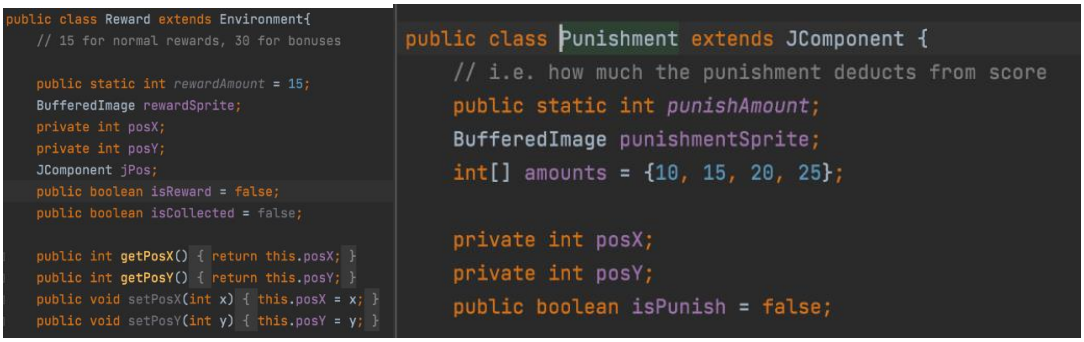

Figure 2: Duplicated code found in the Reward and Punishment classes.

Game objects such as Player, Reward, Bonus, Enemy, and Punishment had many variables and methods in common but were found to be repeated in each class. Some repeated variables include the object's x-position and y-position on the map, the getters and setters, and a boolean for when the object has been 'hit'. Repeated methods include movement validation and randomizing placements on the map. Figure 2 shows Reward and Punishment as examples.

Refused Bequest

As mentioned above, the five game objects were supposedly related to each other and had similar variables and methods. However, each of these classes were subclasses of different superclasses. The previous relationships had no meaning between the superclass and subclass.

```
public class Player extends JComponent {
class Enemy extends JComponent {
public class Bonus extends Reward {
public class Punishment extends JComponent { public class Reward extends Environment{
```

Figure 3: Refused Bequest - superclasses and subclasses unrelated.

These classes have been refactored to inherit from the abstract superclass GameObject. GameObject resolved both the code smells **Duplicated Code** and **Refused Bequest**. The similar variables and methods between subclasses have been removed and added to GameObject so that subclasses may simply inherit them.

The 'Game' class had the issue of being a **Long Class** code smell. The Game class should only be responsible for initializing other major classes such as GamePlay and SpriteManager for rendering and sprite setup. However, it also contained a large set of code to initialize interface and GUI related variables and methods which were completely irrelevant to the Game class and made the class too long. The code smell was refactored by creating a new 'Interface' class which handled all GUI related objects and methods including the start screen, game screen, and finish screens. This refactoring re-organized decluttered Game class from being 151 lines at the start, to just 11 lines after the code review and refactoring process.

Long Methods

Methods in the GamePlay class, such as initializing and setting random rewards, bonuses, punishments; as well as methods that checked if the player's current position had 'hit' a certain object, all contributed to a longer class since each game object had their own method to accomplish the same purpose. For example, the class had setRandReward(), setRandBonus(), setRandPunishments(), checkNearbyReward(), checkNearbyBonus, checkNearbyPunishment(), and checkNearbyEnemy(). Our solution was to refactor each method into one single method where the parameter accepts any GameObject subclass and to perform the same result. As per Figure 4, the previous setRand methods were refactored to share setRandObject(), and the checkNearby methods to share checkObject().

```
public void setRandObject(int n, ArrayList objectList, Class c)
public boolean checkObject (ArrayList <GameObject> o) {
public void resultForObject(GameObject o) {
```

Figure 4: Refactored methods for setting random GameObjects rather than individual methods for each class.