Assignment 3: Code Review Report

To improve the quality of our written code, we began by familiarizing ourselves with various code smells that are often present in poorly written code. We then reviewed our codebase and identified classes that seemed to be poorly designed or difficult to understand. After selecting the problematic classes, we analyzed them in more detail and identified several code smells that needed to be addressed.

We are pleased to report that we have successfully refactored six code smells in our codebase, and have committed and pushed the changes to our group's repository. For ease of access, we added our commits at the end that show some or all the changes that we made to tackle the different smells.

One of the code smells we addressed was low cohesion, which we found in the KeyHandler class. The class contained two methods, handleOverStateInput, and handleWinStateInput, that were very similar in nature and exhibited low cohesion. To improve the design of the code, we introduced a new method called handleEndStateInput that consolidates the common functionality of the original methods. This new method takes an additional parameter, a Runnable, to allow for customization of the "retry" behavior. The resulting code has improved cohesion and is easier to maintain. (85e57307)

We also tackled the code smells of large method and duplicated code in the Sound class. We simplified the class by removing duplicated code and extracting methods to improve readability and maintainability. Specifically, we noticed that the setFile method contained code that was being duplicated in the playMusic and playSE methods. To address this, we created a separate method called loadClip that takes an index and loads the appropriate sound file. We also created a separate method called playClip that takes a Clip object and starts playing it. We left the stop and loop methods unchanged, as they were straightforward. Finally, we removed an empty catch block in the setFile method and added a throws clause to the method signature to propagate any exceptions that may occur. The resulting code is more readable, easier to maintain, and has a more consistent style. (4613ec91)

Another code smell during our code review was that the UI class was too large and trying to do too many things. Specifically, the UI class had many different states, such as the main title, instructions title, level title, and gameplay states making it awkward to change every time the game changed states. To address this code smell, we extended the UI class by subclassing TitleUI and GameUI and then further subclassing those to handle

specific menu states and game states. This helped us simplify the code accessing the UI and made it easier to maintain the UI class. (00f93754)

Creating the different UI classes, allowed us to reduce the number of if/else statements and switch statements being executed while updating the UI. In our GamePanel, the state was originally checked every update in order to correctly set the UI, but now the state only has to check if it was in the title or in the game because the UI object would already be set to the appropriate state object. This smell was also found in the UI class, when an object was picked up the message also needed the object image, which would be passed as an integer and use an if/else statement to match the image. This was refactored by using an array that aligned with the GamePanel object list. Removing these if/else and switch statements were beneficial in decreasing the amount of logic being executed during every execution of draw, which updated the screen every 0.01666 seconds. (1692c278)

There were also instances of code duplication, especially with image generation. We found a separate method for image generation in all of the classes that needed an image, and to solve this we implemented a utility called ImageLoader. With this tool, we also implemented a loadAllImages in our GamePanel which would load all of the images at the beginning of the game and store them in a hash map. This reduced the amount of code in the entity classes, object classes, and UI, and also eliminated the need to load images when they were making the game more efficient. (123e5c97)

We also found unnecessary use of unsafe or unsound constructs in our UI, when writing messages it would create new fonts and colours every time the draw was called. This smell was fixed by creating protected colours and fonts that could be called by any UI subclass. In addition to it being a more efficient and more practical approach, this also created consistency as to what font and colour should be used in what context, for example, TitleFont should only be used for titles. This also allows for easier maintainability, if we ever wanted to change these features. (7d6e4361)

In conclusion, through the process of refactoring our written code, we were able to identify various code smells and successfully refactor them. We addressed classes that were too large and/or tried to do too much, unnecessary if/else or switch/case statements, code duplication, bad/confusing variable names, and unnecessary use of unsafe or unsound constructs. By making these improvements, we were able to create a more readable, maintainable, and consistent codebase. Overall, we were able to improve the design of the code and reduce complexity, leading to better performance and easier future modifications.