CMPT 276 Assignment 4 - Refactoring

Rashed Hadi - Raymond Zhou

Breaking Down the UI Class' Constructor

We chose to break down the UI class' constructor since we realised that it is used to initialise various objects that could each be initialised in their own methods. We believe this occurred over time during the game's development since each team member would end up adding something to the UI, initialising it in the constructor, resulting in a mix of code that belongs to different parts of the game. Before refactoring, the UI constructor contained the the instantiation for the barrel, the oxygen bubbles, and the rewards, which have now been extracted into their own methods called createBarrelObjects, createOxygenObjects, createRewardObjects, which all take in the game panel as a parameter. Finally, the UI constructor has been updated to call these methods so the functionality of the game does not change.

Breaking Down the Main Character Class' Constructor

Similar to what occurred for the UI, we believe that the main characters constructor grew over time which turned it into a long list of instantiations of different elements of the main character. We decided to split them down into new methods that will be called from the constructor instead to improve the class's organisation. We extracted parts of the constructor and then created two methods called setSolidArea, and setBoundaries then proceeded to call those methods from the constructor.

Fixing Inconsistencies in Method and Class Naming

When looking over the project, we realised there are some inconsistencies in the style of method and class naming. This was a result of having different individuals working on the same project, bringing in their own individual coding style. The first naming change we made was the main character class name which was initially Main_Character, and was changed to MainCharacter in all 44 instances across the project. The second change was the method set_default_position in the main character class which was changed to setDefaultPosition. The third change was Reg_Fish, which was changed to RegFish in 19 instances. The fourth was Bonus_Clam which was changed to BonusClam in 6 instances and finally the last change was Reg_Crab which was changed to RegCrab in 22 instances.

Removal of Unnecessary Imports

Scanning the project, we found that more than half the files have some sort of import that is highlighted as never used. This is true for both the source files and test files, so we went through the project and deleted any unnecessary imports. The main import that we found was being flagged as unused is java.io.File, which we are assuming was imported automatically when we were developing the project. We also found one instance of javax.swing.JFrame unused in the testing files along with others such as org.junit.Before which might have been added when creating the tests and ended up being unused when the tests were rearranged or updated.

Breaking Down the Board Generation Methods by Extraction

Focusing on one of the main components of the project, the Board.java class initially had one large function creating and drawing the perimeter of our maze as well as another large function generating the obstacles of the maze. Having a larger method can be a nightmare to debug as it is hard to read and potentially hard to interpret as it looks quite chaotic. To break down the perimeter generation method, we separated the function into two parts. One to generate the sides and the other to generate the top and bottom of the perimeter of the board. In addition, the obstacle generation was also split into 2 separate methods. One is regulating the generation of our stone cell texture and the other one regulates the generation of the seaweed cell texture. Combining all of these individual methods, the generateBoard() method has been refactored to utilise the helper methods to accomplish the task.

Fixing Unnecessarily Long Variable Names

Mentioned above for the Board.java class, this class was a little more challenging to read given the amount of line each method occupies. In addition, some of the variable names are quite long and hard to understand when reviewing the code. An example would be "this.cells.panel.individualTileSize". When this variable is used around other long code statements, it can quickly throw the reader off and cause confusion. These long confusing variables are replaced with local variables that have shorter and more comprehensive names. If one was debugging the refactored code with context to the game, they will quickly understand what the variable "tileSize" means compared to the former. "tileSize" has replaced the former variable name in 74 instances. "this.cells.panel.numColTiles" has been replaced with boardColumns in 11 instances, and "this.cell.panel.numRowTiles" has been replaced with boardRows in 10 instances.

Renaming Class Variables in CellCollection.java to Make Sense

In the CellCollection class, it is responsible for importing and managing all the possible textures in the game. Given that class is the foundation of the board textures, it is necessary to provide clear and comprehensive variable names. This way, anyone who is working on the code can access variables and expect those variables to contain the desired data and not. Specifically, we have an array initially named "cellTextures" that holds all the possible cells of our game based on the amount of textures that were imported. The issue with this is that the array is not necessarily storing cell textures, rather it is storing Cell objects that contain attributes (one of them being the texture type and image). Since the array is really holding the various unique cells, this array's variable name has been changed to "cellType" instead. This is an appropriate name as it hints to anyone reviewing the code that this array contains multiple types of Cell objects, with unique attributes, and not just their texture types or images.

Changing Global Access Modifiers to Private for Encapsulation and Data Protection

In certain fundamental classes, it is important to encapsulate certain variables and change their access modifiers to private or static. This prevents any accidental/unwanted modifications to the data outside of the specific class that the variable was set private or static

to. Assuming no setters are implemented for variables holding data that are not intended to change, this makes debugging much simpler as the issue causing any changes to that private or static variable can only happen within its respective class. As this is a necessary change, the Cell.java class was modified to have all of its variables to be set private and getters were established to make retrieval of the data possible. Since the data in this class is not intended to change throughout the lifetime of the program, no setters were implemented. Moreover, the CellCollection.java class had its cellType array set to private to prevent any unwanted modifications to be made outside of this class as the program regularly utilises this array to classify cells. Again, a setter was implemented to make the retrieval of each individual index of this array possible. Lastly, for the Board.java class, all of its class variables are set to private with the same reason to prevent unwanted modifications to them from outside classes. By changing the access modifiers to some of these key variables, it could be rest assured that the data is protected from any unforeseen modifications.