Ali Simbanegavi & Daniel Cumming

COMP 3721

L. Ricker

November 9, 2020

**Robot Turtles (Milestone 4)**

Our previous design had several SOLID violations and minor bugs. This refactored version addressed these issues by making several significant changes.

Our GameInitializer class’s methods are now static. By doing this, GameController can access GameInitializer’s methods statically without instantiating a new GameInitializer object. This change was made because GameInitializer only needed to be used once to set up the game. By doing so, we were able to avoid creating a new instance of the class that would not be reusable and non-reusable code must be minimized in OOP.

Similarly, GameView’s methods were also made static. 4 new methods and an instance variable for a Scanner were also added to this class. The methods readText() and readDigit() were created so that GameView could read input from the console in addition to displaying information to it. We made this change so that the 2 classes that required input, GameController and GameInitializer, could get input from the console without having to each instantiate separate Scanner objects. Now, a single Scanner object in GameView is responsible for all input. We also added a menu() method to print the initial game menu and put our promptMove() method for requesting player input into GameView. These changes were made so that any functionality related to updating the console view was encapsulated within GameView, which aligns with the Single Responsibility principle.

When starting a new game, our previous version had GameInitializer creating the Turtles, Jewels, and Board, and then passing that Board on to GameModel. In this version, we now only use GameInitializer to create Turtles and Jewels and pass these objects to a new instance of GameModel. This is done by passing lists of the new Turtles and Jewels into GameModel’s constructor, which then calls for a new Board to be built using those objects. This enabled us to eliminate any unnecessary dependency created between GameInitializer and Board. Since GameModel was already storing a copy of Board and Board is a member of GameModel’s package, the responsibility of handling Board’s configuration is now solely GameModel’s.

In the Board class itself, several changes were made. We previously had methods placeTurtles() and placeJewels() in this class, which were responsible for adding each element in the lists of jewels and turtles passed to the game board’s configuration. Now, a single placeTiles() method handles the work of these classes. It does this by creating a collection of all the Jewel and Turtle tiles in the form of a unified list and iterating through that list to place each object at its designated coordinate. Making this change was inspired by the Liskov Substitution principle. With the Jewel and Turtle classes being subclasses of Tile, that means they should be substitutable for their base class. Therefore, creating a method that handles a list of objects that fall under subclasses of Tile, rather than creating a separate method for each subclass, takes advantage of this principle. In addition, when new types of Tiles will be added to a more advanced version of the game, incorporating Liskov’s principle in this fashion has made our adherence to the Open-Close principle stronger. Our Board class is open for extension by allowing new Tile subclasses to be passed to it as input, but partially closed for modification since unique methods to place those new subclasses on the Board will be unnecessary.

The Action class (previously called Move, which was changed to make its function more explicit) has been updated a few different ways. Previously checkForJewel(), which happens before a player is about to move forward, was done by the Action class. Now, the Board class checks this since its main responsibility is containing and relaying the configuration of the Board. In addition, we moved the methods to validate users’ requests to the Action class from GameModel. The responsibility of evaluating an Action’s validity is now part of the Action itself, which takes advantage of the Single Responsibility principle.

Changes to the Coordinate class were made to reduce the responsibilities of Board. Board was previously calculating any changes to Coordinates that needed to happen when Turtles were moving. Now Coordinate can calculate and return a list of all of the Coordinates of its 4 adjacent neighbours. This allowed us to get rid of the coordAhead() and coordBehind() functions in Board by adding getAdj() to the Coordinate class. We can now easily retrieve the position that a Turtle would move to based on the Direction that Turtle is facing and do the same for the position behind a Turtle if a bug card is to be used. Delegating this task to Coordinate made this class less tightly coupled with the Board class.

The Card class now has increased functionality relating to left and right turns. We added a method isTurn() to this class which returns true if the Card being evaluated is a Left or Right Card. This was useful for Action’s execute() and validate() methods, since any move related to a left or right turn would always be executable and this eliminated the need for conditional statements in these methods testing if the Card was Left or Right. There was also an opposite() method added to Card, which returned Left if a Right Card was inputted and vice versa. This function was added to simplify the use of bug cards after a Turtle has turned left or right.

The Direction enumerated type has also been modified to reduce any other class’s responsibilities when it comes to reversing Direction for bug cards, left/right turns. Now, Direction has a reverse() method which can return the opposite Direction to the one currently being faced. This is useful for bug card execution since it can facilitate backwards movement. Previously, Board’s coordAhead()/coordBehind() functions assisted with this. In addition, the new turnDir() method accepts a String input of “left” or “right” and returns the corresponding new Direction if that rotation is to be executed. Previously, Action was responsible for calculating this when a rotation was requested. Adding complex Direction-related operations to Direction itself helped our code adhere to the Single Responsibility principle.

Finally, as a correction to the close coupling of our Model and Controller from Milestone 3, we created a new class in our Model called GamePlay to remove any game logic from the Controller. This class now houses the methods requestMove(), playRound(), playTurn(), and start(), which all contribute to use cases occurring during gameplay. By removing any methods having to do with game logic, our GameController class now exists exclusively to relay data between the View and Model. Having all the pivotal components of the game’s logic contained in the Model has reduced coupling with the Controller classes and strengthened our adherence to the MVC paradigm.

The updated version of our program is a vast improvement compared to our design from Milestone 3. By applying some SOLID principles to our code, especially the Single Responsibility principle, we were able to increase its overall readability and have separated the responsibilities of different modules much more effectively. Our code still has room for improvement, and as we get familiar with new design patterns, our next version of the program will be more flexible and extendable while incorporating new features.