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 for the same reason. 2 new methods and an instance variable for a Scanner were also added to this class. The new 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.

When starting a new game, our previous version had GameInitializer creating the Board, and then passing that 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 Board 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 of 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.

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 should be part of the Action itself if applying 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. 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 further reduced Board’s responsibilities.

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 used and vice versa. This function was added to simplify the use of bug cards after a Turtle has turned left or right.

The Direction class 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 handled this. In addition, turnDir() method which accepts a String input of “left” or “right” and returns the corresponding new Direction if a turn will be executed. Previously, Action was responsible for calculating this when a rotation was requested.

\*\*\*\*\*\*BELOW IS MILESTONE 3, USING FOR REFERENCE TO REWRITE EACH PARAGRAPH

These functions were simpler to implement by using a dedicated Board object during gameplay. This is preferable to diffusing responsibility related to the board layout by putting methods in the GameController and/or GameModel class that were directly responsible for modifying the board in the form of a 2D array.

One thing missing from the original design that we have implemented was a Move class. This class is responsible for handling instructions on cards that would be used by players to move around the board. We found that rather than bloating the controller with commands to execute on player turtles, we could create a reusable class that accepted a player and a move instruction as input. This class worked with the board by making changes to players’ coordinates and the directions their turtles were facing based on receiving an instruction of left, right, or forward. It also processed any bug card requests by reversing moves made by players and applying them to the board. The Move class also checked to see if a player’s target tile for a step forward contained a jewel and relayed this information back to the Board for the player to be marked as a winner.

The creation of the classes mentioned previously was in part because in our original design, the GameModel class was simultaneously bloated and unspecific, as it did not have any clear primary responsibilities. We aimed to reduce any dependencies or unclear methods that that complicated the model’s tasks. It is now primarily responsible for giving feedback to the controller about the adherence of the current board configuration to the game rules. We split this responsibility into 3 simple main subcategories.

The first tenet of the GameModel class’s responsibility was storing and updating the board configuration. The method getBoard() returns a copy of the board in its current state, and updateBoard() computes and stores any changes made to the board. Changes are made after the controller relays data to the model about a move executed by any of the players.

Before the controller can call updateBoard() and request to change the configuration, the GameModel is responsible for validating permissible moves. It did this with the validate() method, which returns a boolean which is true if the execution of a Move can be applied to the Board object. We gave this job to the GameModel to simplify the controller’s execution of the gameplay. This took care of the controller potentially having to handle the unnecessary step of checking if a move was valid within itself before executing it.

The GameModel, in conjunction with Board, is also responsible for keeping track of which players were still taking part in the game. The GameInitializer passes a list of the starting players into the GameModel. This list is what used by the Board class to build the initial configuration. If a valid Move executed by a player results in the player picking up a jewel, the GameModel will update by removing that player from the board and this change will be reflected in the Board class’s list of players.

The changes mentioned previously were all employed to enable the creation of a GameController class that was more specific and comprehensible than what was in our original design. Now, GameController is a class that controls the flow of gameplay, and uses concise methods with help from other classes to do this. It uses promptMove() to request Moves from a player, which calls GameModel to validate these Moves and execute them if they are permissible while re-prompting the user for input if they are not. Valid Move objects are called for execution by playTurn(), which is a method in GameController that processes card commands for the current player. The playRound() method repeatedly calls playTurn() for each player still in the game. playGame() continues to call playRound while the GameModel indicates that the game is not over.

In our original design, it is important to note that we had enumerated types for the game state (ongoing and incomplete) and for the player state (playing or won). We replaced these with boolean values in the GameModel (complete) and player, or TurtleMaster, classes (winner) to simplify checking on the status of each game round and player by avoiding the creation of new types to do this. We also made changes to our Tile class, by creating a subclass from it called ColouredTile. Being the only coloured Tiles on the map, we decided to make the Jewel and Turtle classes extend this class. Furthermore, the TurtleMaster class which represented our players extends the Turtle class. This way, we were able to ensure that player’s representation on the Board could exist as a Turtle with a color and direction, while adding methods for naming each Turtle’s respective player and marking that player as a winner.

In our previous design, our GameView class was contaminated by the use of a Tile[][] array as a parameter for the displayGame() method, which was a member of the model. This constituted a violation of the basic requirements of MVC architecture. We mitigated this in the updated version by creating a BoardConverter class. This class accepts Board as input and parses the tiles within board as 2D int arrays.

GameView now contains 3 essential methods for communicating with user. It can display any text passed to it by the controller via displayText() on an individual line. Another method that serves a similar function is displayPrompt(), which is used when input is requested from the user and allows them to type inline directly after the prompt. displayBoard() is used to display the results of this parsed board to the console. Data from a Board object about the configuration of the board was parsed into 2D int arrays. The lists of Turtles and Jewels obtained from Board were parsed separately. For the Jewel lists, we created an array of with n rows, where n = # of Jewels, and 3 columns, representing each Jewel’s (x, y) coordinate and color respectively. The coordinate and color of each Jewel was passed into its corresponding row. The procedure for parsing the Turtles was similar, except for the number of columns. The first 3 columns represented the same information as that of the Jewel’s 2D int array, but a 4th column was added with an integer representing each Turtle’s direction. Processing information this way, we were able to pass on a complete representation of the board to the view as numerical values and display the board while severing any direct ties to objects that were members of the model.

Overall, significant changes were made from our first design. Upon re-evaluating the structure of the code we had planned to write and observing how many unnecessary dependencies could be eliminated, we extracted several new classes with well-defined responsibilities. These changes made our program more streamlined, reduced the sizes of our methods, and our design now adheres more strictly to the principles of object-oriented programming.