Ali Simbanegavi & Daniel Cumming

COMP 3721

L. Ricker

October 27, 2020

**Robot Turtles (Milestone 3)**

Our initial design had several issues that we addressed in this reconfigured version. Upon learning more about the SOLID principles that we have covered so far and re-assessing the flow of the gameplay, we made several significant changes.

One of the biggest changes we made was extracting a GameInitializer class. We found that to begin the game, several tasks needed to be performed that a single object could handle one time at the program’s start, rather than overfilling the controller with those methods. Giving a single class this responsibility simplifies our code as we were able to avoid filling the controller up with methods related to initializing the game that would only be called once. This class is begins by initializing the view and is now responsible for instantiating the players with createPlayers() and their respective jewels with createJewels().

After these objects are instantiated, a board is created with createBoard() which will populate board with the newly created players and jewels based on their initial coordinates. GameInitializer also constructs a model based on the initial board configuration and number of players. These functions would end up being filler for the remaining duration of gameplay if they were part of the model and controller’s methods. Making a single class to handle them significantly simplified our code.

We decided to simplify the representation of the board by storing it in a newly created Board class. This was preferable to our previous design, where the abstraction of the board was simply a 2D array of tiles. Since the board was subject to consistently changing the information it held and interacting with the model and the view, creating a class with dedicated methods for these functions was more useful than relying on what methods were possible with a 2D array on its own.

Now, Board handles any data that relates to the game board’s configuration by storing a 2D array which represents the layout. It modifies the layout of the tiles with methods shiftTile(), setTileAtPos(), placeTurtles(), placeJewels(), removeTile(), and markWinner(). The Board class also maintains a list of players and jewels currently on the board with getTurtles() and getJewels(). The methods getTileAtPos(), isEmpty(), and nextCoord() also handle information about the current layout.

These functions were more simple 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.