Design Decisions and Data Structure Justification

Ryan Boucher

December 4, 2018

1 Main Changes in Code from Milestone 3

The main changes in our codebase from Milestone 2 is as follows:

1. Introduction of a real-time aspect to the game
2. The addition of a level-builder
3. The addition of save and load features
4. Fixing of the undo/redo feature, as well as sun-point updating on undo/redo
5. Additional Testing

2 Real Time Plants VS Zombies

From a user perspective, the largest change in our codebase for this milestone involves the transformation of our game from a turn-based strategy game, to a real-time strategy game. In other words, the zombies will no longer wait for the user to make a move before they themselves making a move, but will act in real time to attempt to defeat the user.

To achieve this, our group has introduced a timing system, which when implemented in our controller class, triggers the runtime() methods, moving our game along, regardless of whether or not the user has made a move.

This is a two-part process: The use of a Timer, and the use of a TimerTask; both of which are in the java libraries.

TimerTask: This object, essentially, holds code that can be executed by a Timer object. In our case, our TimerTask has a brief delay, before checking if the level is over (cehcking the level of the game and the nunber of zombies), and if the level is not over, executes our runtime() methods, moving the zombies.

Timer: This object, executes after a certain amount of specified time has passed, and calls its TimerTask, which has been specified above. This largely controlls how fast the game runs. Changing the value passed into this timer can **dramatically** affect the speed and difficulty of our game, as it designates how fast the computer-controlled zombies can act.

3 The Level Builder

Using the class XMLParser, we have added for the ability of a level-builder into our codebase. By specifiying details in our XML file, GameLevelFile.xml, and then feeding it into our XMLParser, custom maps can effectively be created. A variety of options can be chosen, ranging from difficulty options (such as total zombie count, zombie limit, and zombies spawned), to player advantages (amount of starting sun-money, max level, etc.).

Much of the functionality of the level-builder is tied into the additional class that has been added into our model, called GameLevel.

4 Save and Load Features

Using serialization, we have added the ability for the user to save and load their game, which is stored as GameSave.txt.

Using a save() method, the code (implementing serializable), writes the data of the board into the text file. Using the load() method, we assign a temporary board to the data held in the GameSave.txt file, and then copy that board into the current gameboard.

5 Undo/Redo feature bug Fixing

As a noted problem in our last milestone, there was an error in our undo/redo functionality, in which the very first move that the user made could not be undone (and as such, could not be re-done). This is a simple fix, as our undo/redo functionality operates through the use of a stack, and the contructor dealing with the creation of the board did not push the initial state of the board on said stack.

With the contructor pushing the intial state of the board into the undo stack, the undo/redo feature now works properly, and does not have any known errors.

6 Continuation of Rigourous Testing

As with the last milestone, the testing our of our code base has been updated to properly test all added features.

Notably, our GameLevelsTest in our test sub-directory tests all gameplay that can be tested, when using our level builder and/or save and load functionality.

7 Data Structures in this Milestone (Milestone 4)

With regards to what has been added during this milestone, no new major data-structures have been added into the code that haven't already been discussed below.

Our undo/redo has been fixed, and that has been accomplished by the addition of another stack (of which holds Integer SunPoints). This has not fundamentally been a change in logic as to what was described in our Milestone 3 changes (of which the undo/redo stack is discussed).

Additionally, the use of XML and serialization does not necessarily constitute a discussion of data-strucutres, although it is worth mentioning that they have been used, as is discussed above.

8 Main Data-Structure Changes from Milestone 2

The biggest change in our codebase for this milestone was the addition of an undo/redo functionality into the program.

To achieve this functionality, first, a "Deep Copy" of the data of the board had to be achieved, in order to allow for the state of the board to be written into memory correctly. To achieve this, at the end of every player turn, the game must write the contents of our board, consisting of squares, that consist of pieces, into memory. As such, we have added copy() methods into class Square, as well as every one of our Piece subclasses. These copy methods generate **new objects** that are proper copies of the current state of the board. We copy every square of the board into a new board object (as well as the pieces), and then proceed to store this information into an **Undo Stack**.

When the user clicks the undo button, the undo stack pops, and sets the state of the board to the popped board. Additionally, when this stack is popped, the object is pushed into another stack, the **Redo Stack**. For every sequential undo that is triggered, the redo stack will grow in size. When the redo functonality is called, the redo stack will pop its board object, set the current status of the board to the popped board, and then proceed to push the popped board into the undo stack.

When the user places a piece after performing an undo, the redo stack is wiped.

This functionality allows for a user to perform an undo operation until there are no more possible undos, but, also allow to perform a redo, if they are unhappy with the undo.

Other main changes in this Milestone include:

1. Improved graphics, in the form of grass, and of more plant icons
2. The addition of many more pieces into the game

3. The refactoring of some of our controller class code, to allow for increased readability

4. Complete testing for all new changes, and expanded testing into our classes

9 Java Class: Controller

This class functions as the controller for the project. Additionally, it models the board that the Plants VS Zombies game runs on, and was previously called GameBoard. It holds information on the amount of money that the user currently has, as well as the limit on the number of zombies that will occur on the board (a rough version of difficulty).

The board in this class has been modelled as a two-dimensional array, to allow for the use of rows, and columns. Two dimensional arrays are fast in regards to their access of information, and as the board is small (8 columns by 5 rows), the game has lost little in regards to efficency when it iterates over the board.

This class has been expanded to allow for the use of an undo/redo function, as discussed above.

As a controller, this class assigns an ActionListener to each square of the board, and listens for the user to click on the popup to place a plant on the board, before executing the appropriate code.

10 Java Class: View

This class functions as the view for this project, also known as the GUI. It renders the board, plants, and zombies that the player will play the game on, and is responsible only for rendering. It renders based off of the current state of the board, which is controlled by the Controller class.

There are no significant data-structures to discuss in this class.

1. Coordinate

2. Peashooter

3. Piece

4. PlantPieces

5. Square

6. Sunflower

7. Zombie

11 Java Class: PlantPieces

This class allows for text-based representation of the pieces on that will be used in the game. This is primarily used for milestone one, in which we use a text-based display of the basics of the game.

There are no significant data-structures to discuss in this class.

12 Java Class: Piece

This class models pieces used in this game, and includes information on individual piece health, damage, cost, and both the name, and short-name (char) that is used to represent the piece. Pieces will exists as objects in this design, and as such, it makes logical sense to model them here.

There are no significant data-structures to discuss in this class.

13 Java Class: Coordinate

This class models coordinates that will be used to identify locations on the board, through columns (X values) and rows (Y values). This is necessary to allow for easy identification of where pieces are in the game.

There are no significant data-structures to discuss in this class.

14 Java Class: Square

This class models the individual squares that make up the game board. It is used when adding and removing pieces, and will identify if a square is currently being occupied or not. This class contains the logic for adding and removing pieces on the board, and is integral for the game.

There are no significant data-structures to discuss in this class.

15 Java Class: Main

This class intializes the MVC modelling used for this game, and does the intial calls to View (the GUI) and Controller (the controller).

There are no significant data-structures to discuss in this class.