**Design Decisions**

For our Plants vs Zombies project, there were numerous decisions that we had to make to decide how we wanted to implement something, and if we wanted to keep or remove certain features. This document will go over our thought process behind our design decisions.

**Model**

Originally, we planned on using a 2D arraylist to simulate the board, however we decided to go with a nested arraylist, with a custom cell object contained in the inner arraylist. Both serve the same purpose and it did not matter which data structure we used. However, for the list of zombies and plants we went with a LinkedList because there would be more removing and adding to the list opposed to a need for indexing the list.

We decided that the redo and undo buttons should save the board state to a stack and every move will push to the undo stack. This was the easiest method and made the most sense since undoing a move reverts the board by the most recent pushed state, so we wanted a data type with FILO structure. The redo stack would get cleared every time undo would get pushed to so that you cannot redo an older move you made.

For save and load we didn't use XML simply because it seemed much easier to just use a separate stack and do the same logic as undo/redo. Custom levels currently only consist of modifying how many zombies will spawn. More ways to customize the level would be to modify the dimensions of the board, as well as choosing which plants to bring with you. Choosing plants would require more variety in plants, however.

There were a limiting number of plants that we could create based on the original game. The original game had different levels that would alter the board, such as night time, the pool (normal plants couldn't be placed in the water), and varying elevations over the board. Some plants had other uses by changing the projectile type, such as making it a sharp needle, or catapulting the projectile overhead. Our zombies are not special in any way, and we do not have various board maps, so none of those plants could be made. To keep things simple, we did not create a projectile object, and simply had the plants scan their row and damage the first zombie they encountered.

One design implementation change is that placing a plant down onto a grid that already has a plant will replace the plant rather than prevent you from planting it there. That is placing a sunflower where a peashooter is will remove the peashooter and plant the sunflower there. The original game did not let you plant where another plant already was. If you wanted to change the plant, you could use a shovel to remove the plant before planting it there. We decided to not create the shovel and instead simply had the plant replace one another.

The game has not been balanced to be difficult or progressively hard. It is simply a basic skeleton of Plants vs Zombies with working features.

**View**

View was done well in the end, all functions worked as intended when testing. I ended up getting the undo and redo button fixed within 24 hours after Milestone 3 was due which was unfortunate. The game visually looks great, but we couldn’t add in zombie sprites in time. Another visual improvement would be to have the input for number of zombies that should spawn appear after the game appears, rather than only the input window appearing alone.

**Controller**

Controller works as intended and well. Things that could've been done differently are using a different method of differentiating between the grid and other buttons. Currently if a button in the View isn't in the grid, we set it to the normally out-of-bounds index of 10,10. If the buttons index isn't 10,10 then we know that a plant is not being placed, so we should call a different method. The alternative way of doing it would be adding the index of each grid button to the button itself as a name, and checking that instead of checking the grid.

We made a new object GridCell that serves the same purpose as coordinates from the TicTacToe example, but we made the GridCell able to hold a single plant and any number of zombies. Rows and columns of gridcells would then accurately simulate a board. We needed to create an object that could hold multiple objects, and the GridCell accomplishes this.

**Remaining Decisions**

The game uses the MVC structure like the TicTacToe example. The view is first created and creates a new model. Every button in the view has a controller to monitor it. When a button is clicked, the controller calls the according method in model to change the model’s state. The model then creates a new event and status and sends it to view. View then updates its visuals to match the updated board state.

The statuses we have are placing/removing a plant and zombie, the player has won or lost, and updating sunlight. These were decided by going through all cases, brainstorming when a visual update to view needed to happen, and grouping the statuses that were similar.

There were limited number of ways to customize the levels. One would be to get user input to choose how many zombies to spawn. A second would be to adjust the stats of the zombies. We also considered allowing the user to adjust the board dimensions, but we were not able to get that working in time. An alternative but similar custom adjustment to the stats would be to allow the user to choose difficulties, which would automatically adjust the zombie’s stats accordingly, such as health, damage, and speed. We decided to go with manually receiving input as that was simpler to implement.

**Kevin’s Reflection**

The code can be difficult to read and interpret from an outsider’s perspective. However, I believe that most of the code is necessary and there is not much simplification that can be done without major refactoring of design choices. I understand that the game needed to implement MVC design because that is part of the course, but it should also be understood when and why MVC should be used in certain cases, and when it shouldn’t be used. Would having the game incorporate MVC overcomplicate things, or does it accomplish exactly what is needed?

Overall, I liked the project because it is more enjoyable to create a game than it is to make a general use program that has no purpose in-real life. The game may not be a shippable product, but it gives everyone an idea of what logic is needed to build a game, and how easy or difficult some components can be.

**Leo’s Reflection**

I liked how the specifications of the project made us implement an MVC design. MVC design is found everywhere in the real-world applications although there are many tweaks and work arounds that don’t always follow a strict MVC template. MVC is still important in keeping code modular and decoupling.

Although implementing a game is straight forward to follow the MVC concept, I think there should be more options for either a different game to implement or a different program altogether to allow for students to have more options and work on something they might be more interested in while still implementing MVC. With more options it would also help lead to the fourth-year engineering project which has way more freedom for working on anything of the student’s choosing.