Design Manual for Monsters Vs Plumbers

This program is modeled using a modified version of the model-view-controller (MVC) design pattern. The major difference between standard MVC and our structure is that since we used Scene Builder to create the static view, our controller had a much more involved role in maintaining the view than it normally would. This program uses 4 different screens that are all stored in the same window on top of one another. Whenever screens are not needed they are simply made invisible both visually and to the mouse. For example, beginning the game simply deactivates the menu screen and activates the game screen. All aspects of the game except for the center section of the game screen's BorderPane were built in JavaFX Scene Builder and then connected to our view controller, the *TowerDefenseUIController* class*,* while that specific section is generated via code.

The game loop is controlled in the handle function of this class and tells the game to update after every 'tick' where a tick is defined as 1/60 of a second. We use ticks to determine when and how often certain events should take place. For example, an enemy's movement speed is defined in terms of movement per tick.

In terms of the object oriented structure of our code, we treated the game itself as an object to update with every tick. The game contains a *Board* that represents the gridlike game screen that a player sees. The board is made up of *TileRows*, and each *TileRow* is made up of individual *Tiles.* Within each tile, the program checks for collisions between instances of the *Tower*, *Enemy*, and *Projectile* classes. We use this process of decentralized collision detection so that the program does not waste processing power checking for collisions between game objects that we know for a fact are not colliding at that time. For example, if an enemy is 2 tiles away from a tower, a collision is not possible, and thus the game should not be checking for said collision.

Towers are placed on *Tiles* by the user. Each tower shoots projectiles at enemies, each having its own firing speed and damage dealt. There also exists a tower that acts solely as a barricade by launching non-existent projectiles once per year that the program runs. The user is able to buy towers with the money they acquire over time.

The amount of money the player has at any point is calculated by the *MoneyHandler* class, which credits the player money each second and additionally whenever an enemy is killed, and debits money when the player purchases a tower. This money value is displayed in the top right of the screen via a binding to one of the labels in the *TowerDefenseUIController.* Similarly, the amount of time the player has survived for displayed below it. This value is handled by an instance of the *SurvivalTimer* class.

The user is also able to decide how difficult they want the game to be. The level of difficulty is handled in the enumerated type *Difficulty*. The more challenging a difficulty level selected, the more expensive towers cost for the player. In easy mode towers cost 25% less than as they do in medium mode, while in hard mode towers cost 25% more. The actual purchasing and placing of towers is done entirely by mouse click. They first click on the type of tower they wish to place, which is stored in the *TowerDefeseGame* instance as the actively selected tower. The user then clicks on the Tile in which they want to place the tower, causing the program to the place whatever the actively selected tower is in that location.

Our user stories were as follows:

Complete:

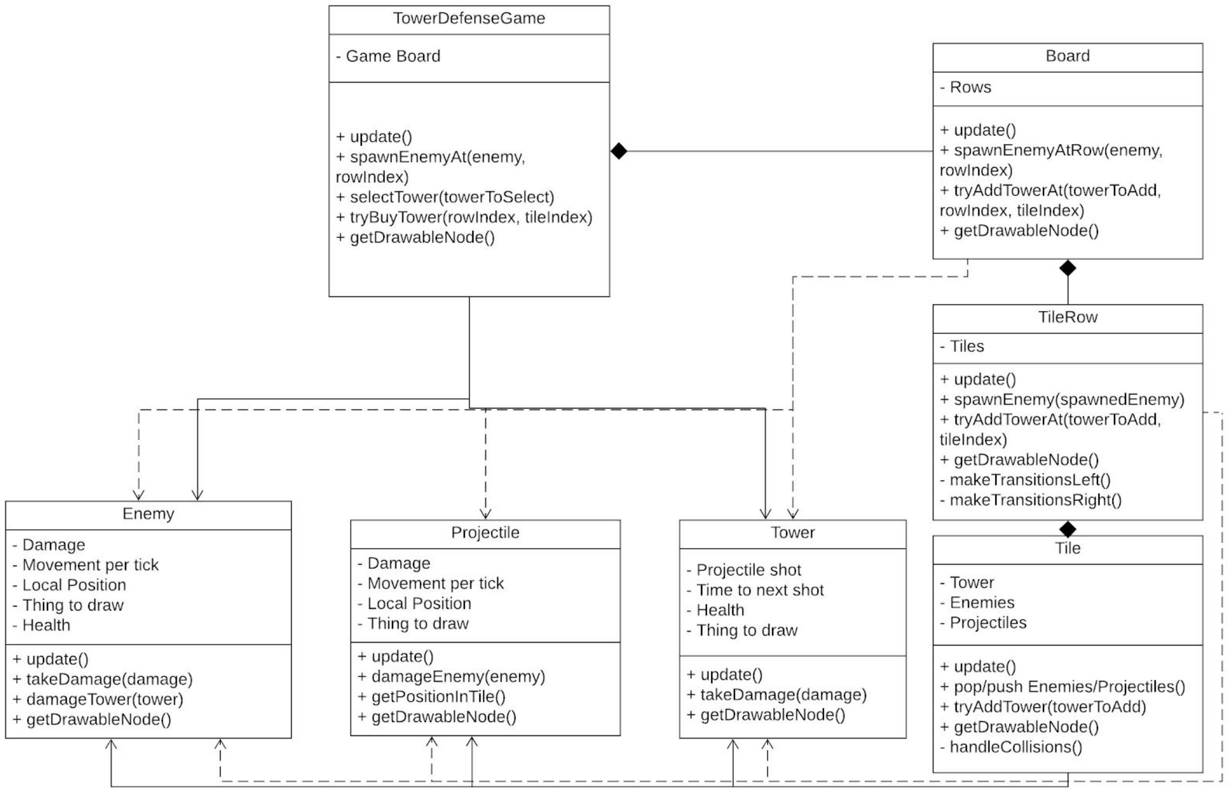
- Start the game with main menu with “play” and “options” buttons
- Be able to change the difficulty to customize the experience
- See the background (board) with tiles clearly shown
- Interact with a bar at the top of the screen that displays the user’s score, money, and available towers
- Place towers down on a board
- See towers pop up when placed
- Have the towers damage and eliminate the enemies
- Have multiple types of towers and enemies to interact with
- Be able to pause the game
- Be able to exit the game
- See enemies spawning on the right, moving left
- See enemies destroy towers upon interaction
- Experience a game over when enemies reach the left side of the screen)
- See enemies in ways that reflect the game's theme
- Fight enemies that spawn in a way that makes the game more challenging over time

Incomplete:

- Sell towers back for some fraction of their original cost
- See the stats for the tower at the bottom of the board when having selected it
- See semi-transparent preview of tower when hovering over tile
- Be able to place different towers with unique special effects (e.g. slowing enemies down)
- Be able to face stronger enemies after the game progresses for a certain amount of time

Fortunately, we were able to complete the majority of our user stories. Unfortunately, getting the basic game up and running took longer than expected, so some of the more fun-oriented features (like guaranteeing that you *will* lose, given long enough by creating consistently stronger enemies over time) were incomplete by the end of the development period. That being said, we are still quite happy with where we ended up, and we have a very solid design as a framework on which our program is built, even if there are some pieces that aren't as elegant as we might wish.

For the design, there were two main components. We used a slightly modified version of the MVC design pattern for the broad design, and there was a clear structure for interaction with the game components through the board (depicted below).



The pictured dependencies are largely irrelevant to the general design, but they do reflect its connected nature. Essentially, as we described in the beginning of the document, we tried to localize everything to as small a unit as possible in the program as possible. The CRC cards below for the board structural classes demonstrate the initial thoughts for the interactions between the classes:

Class: TowerDefenseGame	
Responsibilities: *Knows and displays score, previewed tower stats, money* *Knows difficulty (cost multiplier)* Manages time system (constant runs/sec) Runs gameplay loop through board Validates buying of towers in upper UI Creates and passes enemy spawns to rows through board	**Collaborators:** Board Tower Enemy

Class: Board	
Responsibilities: *Knows all rows that make up board* Iterates time step update over rows Delegates addition of towers and spawning of enemies to appropriate rows	**Collaborators:** TileRow

Class: TileRow	
Responsibilities: *Knows number of enemies who have passed the end* *Knows tiles in row* Handles transitions of enemies and projectiles between tiles Handles spawning of enemies in last tile Handles enemies reaching end of row Iterates time step update over tiles	**Collaborators:** Tile

Class: Tile	
Responsibilities: *Knows the tower, projectiles, and enemies on it* Iterates time step update over enemies, projectiles, and towers Handles collision detection between projectiles, enemies, and towers Facilitates impact of collision between projectiles, enemies, and towers Spawns projectiles from towers	**Collaborators:** Tower Enemy Projectile

The goal of this design was to minimize frivolous calculations and make sure that we did not have an implicit structure that we were simply being too lazy to implement in a proper object-oriented fashion. This led to creation of structures like the *Board* class. You could say that the game has a board, a list of rows of tiles, or a 2D-array of tiles. Each of those is an interpretation for how a game could be implemented with decreasing abstraction. However, since this project was intended to focus on proper OOD, we decided to maximize abstraction even if it ended up having a somewhat frivolous construction in the form of the Board class. The game has larger responsibilities than the board, however, which is reflected in the helper classes of *SurvivalTimer*, *MoneyHandler*, and *EnemySpawner*. None of these classes were strictly necessary, but they helped to keep different functionalities of the game class distinct, which assisted us in both development and understanding.

The quasi-MVC style was created due to the structure that JavaFX implicitly provides. While JavaFX has bindings that allow a very strict separation of model and controller, using bindings in a program implicitly requires exposing the internal variables of the program to external comprehension and modification, which is not consistent with what we learned as good object-oriented design. Therefore, we decided that it made the most sense to have the objects only pass down their information as the visual medium as which they would ultimately be displayed. Instead of having our controller manage all of the different aspects of how to display the game board, we distributed that responsibility to the different components of the game itself. Therefore, we have that a *Tile* object is responsible for display it and all contained *Enemy*, *Projectile*, and *Tower* objects, each of which is responsible for their own representation. That idea extends down the board, with the *TileRow* working from the provided displays of its *Tiles*, and the *Board* working from the provided displays of its *TileRows*. This allows the program to create a connected and informed display without exposing its internal components to the controller or other classes. Additionally, it allows for a great deal of flexibility for changing the display system - this structure made it incredibly straightforward to change our display from being a simple shapes-based display to one focused around images because we left ourselves that freedom with our polymorphic setup. Overall, we put a lot of effort into ensuring that we did not get lazy and lose cohesion by abusing accessors and mutators; the only accessors present in the program are for the helper properties and for properties that are associated with objects but are not used inside of the objects themselves, like an enemy’s kill bonus or a tower’s cost. This gives us an incredibly tight and cohesive design with specific and directed functionality present in each class.