Team Name:

Yebin-Jamie-Edwy-Ryan

Team Members:

Yebin Brandt

Jamie Sangerman

Edwy Cai

Ryan Hittleman

Our team implemented a tile-based map. Tiles and their content were separated to allow future implementations of content that span multiple tiles. Map paths are auto-generated using random endpoints along a predetermined pattern of turns. These patterns are specific to the difficulty selected for the game. Intermediate and hard maps have more than one path for enemies to travel on to present a more difficult gameplay. Most of the fields are kept in GameMap, an abstract class, and Map\_Easy, Map\_Intermediate, and Map\_Hard inherit from it, because they share a lot of fields, yet have some differences.

Players can buy towers with gold, and obtain more during game-play by defeating enemies. There is a player statistics display on the top of the game screen. It shows player’s health, gold, score, and number of towers and enemies. Instead of showing health as a number like the other fields, we decided to implement a health bar using ProgressBar to resemble an actual game. We used Observable and Observer so that these stats can be rewritten every time there is an update to the player. Also, the map, enemies, and towers are redrawn every time a change has been made by Platform.runLater().

We have a MainView where the player can read the rules of the game before game-play, load an existing game, and choose from three maps: easy, intermediate, and hard. For these inputs, we decided to use buttons and button handlers. When game rule button is clicked, another window pops up, so that the MainView remains unchanged and the player can exit out of it anytime. Save game option is in the game screen called GameView, so that player can save the game anytime during the game. For save and loader part, we decide to use an txt file to save all data as String and when we load it, we convert data from string to object again. In saving as txt file, we have to use many loops to write LinkedList like Enemy, Tower or Map. Sometimes, we will change int to double, then we have to adjust our saving style and convert it when we need. Win/lose message is shown on the game screen when the game ends.

Enemies continuously emerge from a specific location we call “spawner,” and they move towards the destination that the player will defend to stay alive, which is the “destroyer.” The start wave button spawns these enemies from the spawner and starts the game loop when clicked. For the game loop, we originally used a while loop and sleep(). However we found it problematic so we used a timer which utilized a timer task making each tower attack and each enemy move once while checking for dead and escaped enemies. The timer task runs a single iteration of the core game loop scheduled by the timer to run every 250 ms, however, the game ran at 500 ms per cycle on normal speed. The player may pause and resume the game during the game loop, and also toggle the speed of the game between a normal speed and a fast speed. A cool implementation was the way fast and normal speed was implemented. The timer would skip every other iteration of the timer task if fast mode was off to emulate a normal game cycle speed of 500 ms. If fast mode was turned on, it would simply not skip any iterations but would run at 250 ms a cycle. By doing this, a game speed of 500 ms for normal and 250 ms on fast was achieved.

For the movement of enemies, we used animated gifs and kept the enemy directions in a list. There are three types of enemies, and each enemy has their own speed, armor, and health, so some are faster, some have more armor, and some have more health. We implemented this by having the getters for those fields call its parent’s getters and add some amount to it. Every enemy should have they current health and original health, then we can make the health bar on GUI, for easily understand we use setHealth() method instead of adjustHealth(), because we don’t want to make math way inside the enemy class.

There are three towers to choose from. Their characteristics are speed, range, and damage. Each tower has a different way of attacking enemies we call special powers; they include poisoning, freezing, and slowing down the enemies. For a short period of time, poisoned enemies lose certain amount of health, frozen enemies are stunned, and slowed enemies’ movement speed is reduced. We implemented this by setting their status back to normal after a couple of moves. The effected enemies will have transparent gifs drawn on top of them.

Towers are clickable, so the player can view its statistics and range and upgrade them if needed. They are also draggable so that they can be moved around, and placed on map when buying. We used several handlers that use MouseEvent and DragEvent. The tower images from shop were ImageView instances so they are displayed by adding them to the VBox and setting them on border pane. This made it easy for them to be draggable, since EventHandler methods can be called on ImageViews. However, the towers that are on the map were drawn on canvas by drawImage(), which made them Image instances, so they had to be implemented differently. We had to make the the whole canvas clickable/draggable for moving towers; if the clicked or starting drag position’s tile contained a tower, we set the drag view as the tower contained in tile to make it seem like you’re actually dragging the tower instead of the whole canvas. To differentiate the shop/move drags, we kept a variable to determine if player is shopping for towers or moving, and set them true/false when drag is detected. If drag is detected from the ImageViews from shop, it’s set to true for shopping, and if drag is detected from canvas, it’s set to false which means player is moving the tower.

For upgrades, player is given a choice to upgrade one of the following: speed, damage, and range, and towers can only be upgraded twice, so the maximum version of tower is 3. Upgrades change the look of the tower as well. We used a regular button for the upgrade, and a set of radio buttons for the upgrading options. If no radio button is selected, nothing will be upgraded. We used radio buttons so that the user can only choose one option for the upgrade. And the upgrade option is only displayed when user clicks on the tower, because it made more sense that way.

Tower attack animation shows the tower shooting upwards, and its attacks show animated explosions on enemies. We decided to use gifs for these to make it simpler. We used a boolean variable to keep track of enemies that were attacked by tower, and drew the gif using drawImage() when the variable was set to true.

The player loses the game when player’s health goes down to 0. Enemies are designed to do the amount of damage to player when it reaches destination depending on their remaining health. The player can win the game by surviving 5 rounds, each of which spawn more enemies with the increasing round number.

Before the first team meeting, we selected six design patterns that we thought would be appropriate for this project, and they are iterator/iterable, observer and observable, MVC, template, decorator, and composite design patterns.

We used four of the six design patterns that were discussed: iterator, observer-observable, model-view/controller, and template design patterns. We used the iterator pattern, because we wanted to traverse through the elements in the collection such as the enemies or towers sequentially while modifying them. To update the GUI every time a model object has changed, we had to use observer-observable pattern to notify that there has been a changed made by user input, so that enemies and towers can be displayed on the map accordingly. For the MVC pattern, it seemed reasonable to separate the classes that have different functions; model from view and controller. All the data are held in model, views are for the display of the GUI, and controller handles the user input. The template pattern was necessary, because our map, enemy, tower each had 3 different types. All had variables that they shared, but each type needed to have something special about them, so we thought that it would be best if we had an abstract class for map, tower, and enemy, and have subclasses that inherit from them.

The remaining two were determined to not be appropriate, and they were decorator and composite. At first, we had a regular map class, an intermediate map class which extended that class, and a hard map class that extended intermediate. They have @Override methods such as setUp(), and as the difficulty level went up, the classes had additional members. However, it seemed more reasonable to just use the template design pattern like towers and enemies to keep the consistency with the model design. And the composite design pattern could have possibly been used for the waves of enemies, by creating a class called Wave, and that wave can contain one or more waves of enemies, so that we can spawn different numbers of enemies each time. However, we did not use that design pattern, because we deemed that it was unnecessary. A simple list collection of enemies did the job, and timer/timer task handled spawning the different number of enemies pretty well.