Team GSJ

Grace Cook, Sam Santomartino, John O'Brien

CSCI 205 Final Project

Design Manual

The basis of the implementation of this project was using the Model View Controller design pattern. Also, object oriented programming was utilized in the structure of the classes. Most of the objects involved in the game were created using objects, such as the frog, the moving objects (cars, logs, and turtles), the road, the river, and the lily pads.

Because we implemented this program using scrum, the completion of the functionality was entirely dependent on accomplishing user stories. The following user stories were completed for this project:

- **As a user, I want a background so that my game has a base to run on**.
- **As a user, I want to move the Frog so that I can make it to the other side.**

This user story references the preliminary functionality of the Frog object. In this case, the frog started as a single rectangle. The rectangle was created and shown to the screen. Then, the ability to move was given to the rectangle, using the arrow keys. When the arrow key is pressed once, the frog moves once in that direction. Additionally, the frog(at this time just a rectangle) was bounded. This means that it was not able to move off of the screen. Because the game was only in its preliminary stages, the frog was not its own class yet.

- **As a user, I want cars to move across the screen so that there is something to avoid when moving across the road.**

The first thing that was completed for this user story was the creation of the Car class. Then, the cars were given the ability to move by adding in a path and path transition for each car. Finally, multiple cars were added to the screen. Also, when the frog collides with a car, it moves to the bottom of the screen in order to "restart." This is where threading was first introduced into the project. Each lane of a car creates a thread and checks for a collision with the frog. This implementation of threading later created issues in other areas of the game, and therefore was revised to be in control of the Frog, rather than the cars (and later water objects) themselves. Only when the Frog moves up or down a level is a CarCollisionTask or WaterObjectCollisionTask created and checked. This way, the minimum amount of threads are created and they do not interrupt or collide with each other, making for a much smoother game interface.

- **As a user, I want the frog to look like a frog so that it is more realistic.**
- **As a user, I want the frog to only have to worry about cars in one lane.**
- **As a user, I want the cars to be spaced well.**
- **As a user, I want logs and turtles to move across the river so that my frog has a path to safely cross.**

The first task for this user story was to create the WaterObject class, which represents logs and turtles. These water objects also extend ImageView so that they can realistically look like logs or turtles. The movement of these objects is very similar to that of the cars. However, the difference is that the frog is safe when it collides with a water object. Therefore, in the FroggerController class, we created a collision checker that resulted in a safe frog if it is on the water object and not safe (loss of life) if it is on the river but not on a water object at the same time. In order to implement the movement of the frog with the log or turtle when it lands on it,

threading was used that checks the position of that water object and updates the position of the frog accordingly. The thread is terminated when the user clicks either the “UP” or “DOWN” arrow key, and the user regains control of the Frog’s position on the screen.

- **As a user, I want to have multiple lives so that it isn’t too difficult.**

For this user story, the concept of multiple lives was implemented. In the bottom left hand corner of the pane there are heart images representing the number of lives the user has left. There is a counter in the program that decreases by 1 each time the frog hits a car or the river. When this counter runs out (and the number of lives shown on the screen is 0) then an exit screen appears. In the FroggerView, all of the objects contained by the root are cleared, and the exit information appears. The option for the user to “Play Again” is now displayed as a button, and another game can begin again.

- **As a user, I want a high scores screen so that I can compare with my friends.**

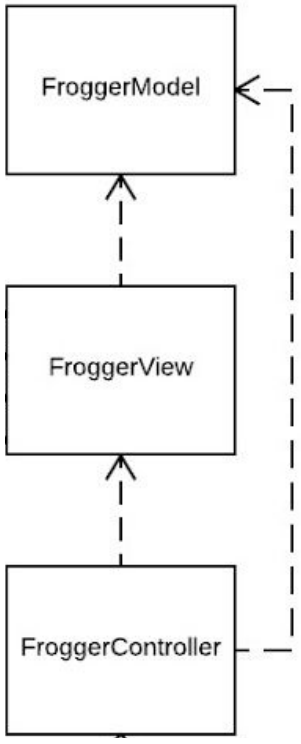
First, we had to add in a counter that kept track of the user’s score. Then, we had to add in a way to compare this to previous players in the past. In order to do so, we created a file that contains the top 10 scores. When the top scores need to be displayed, they are read in from this file. Then, if the user’s final score makes the list, it is added and then that new high scores list is written to that same file. The storage of the high scores is done using a file that is read in and written to. These scores are displayed to the user on the exit screen.

- **As a user, I want a start screen so that I know when it will begin and which level I am choosing.**

The start screen has the options for “Beginner” or “Expert” mode. After the “Start” button is pressed, the user option is sent into the controller to start the moving object Threads

with the appropriate time delays between each object, depending on the choice. Beginner mode has a 3 second base delay between each object while Expert mode has a 1 second base delay, making it much faster/harder to navigate. While the Objects are making their first transition across the screen behind the scenes, the progress bar on the start screen is “loading” to indicate that the game is preparing for play.

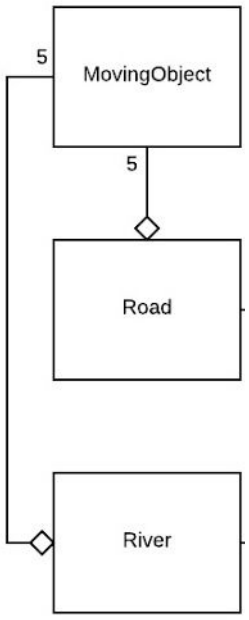
After completing all of these user stories, the game was finished. The main design strategy used for this project is the Model View Controller. Because this is a graphical game, most of the implementation occurs in the view and controller. There are 4 main classes used for implementation and those are FroggerMain, FroggerController, FroggerView, and FroggerModel. The relationship between the controller, model, and view is as shown below in a small piece of the UML class diagram:



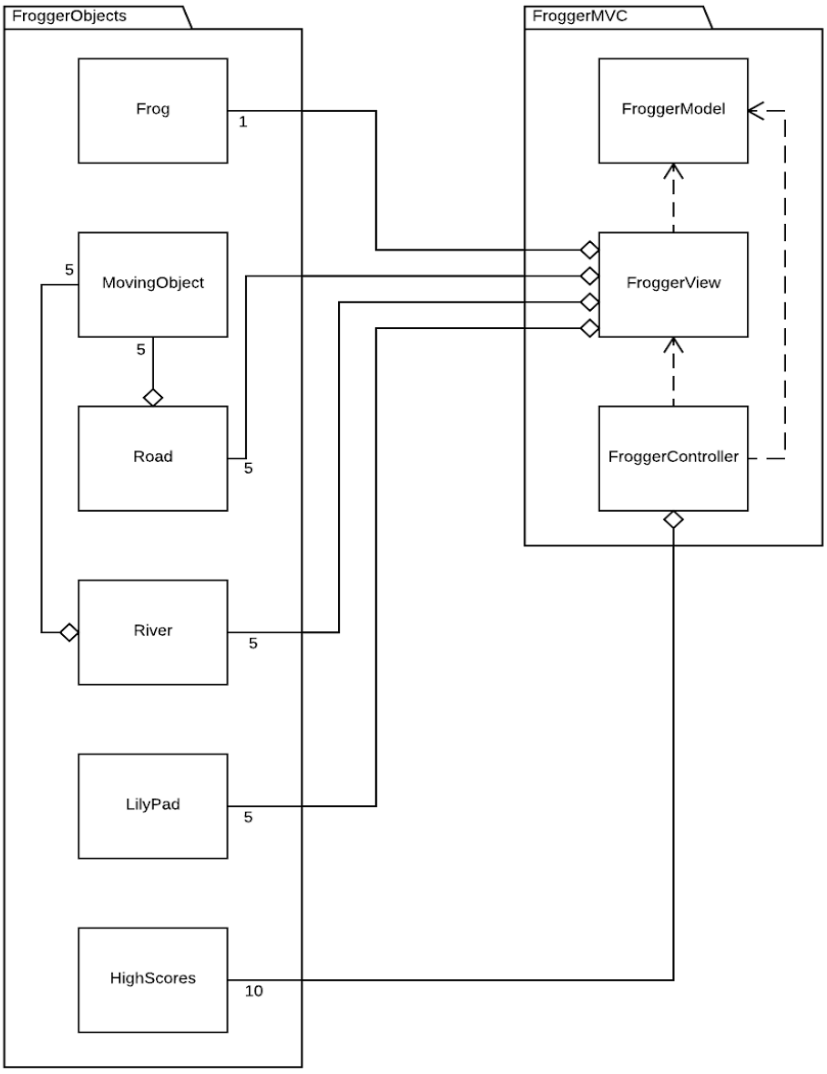
The FroggerMain creates the controller, view, and model. It also sets up the scene and shows the GUI to the user. The FroggerModel does some of the work that is separate from the GUI and that isn't necessarily directly shown to the user. This class contains the back-end calculations. For example, it calculates the starting position of each river and road. The FroggerView creates many aspects of the GUI. For example, the background image is created in FroggerView and added to the root. The view also has methods that place objects on the screen or take them away. Also, the visual objects' design details (ie. size, color, spacing) are established in the view. The FroggerController class does a huge bulk of the work for the game. This is because it controls the connection between the user's actions and the display to the user . For example, when the user moves the frog into the same spot as a car, the controller handles the results of this. In this case, the frog is restarted (sent to bottom of screen), a life is lost, and a heart is taken off of the screen to symbolize the loss of life. In cases such as this one, the controller is the class that checks for collisions such as this one. Then, if a collision is found it calls to the view in order to remove a heart from the screen which symbolizes a life being lost. Also, another method is called in the Frog class that starts the frog back at the bottom of the screen. The controller brings together the objects such as the frog and the moving objects with the movement and visuals on the screen. These are the four main classes for the functionality of the program that use the model view controller design pattern.

In addition to the MVC that is used, we also utilized object oriented design. There are multiple other classes used that represent objects in the program. These include: Frog, MovingObject, River, Road, LilyPad, and HighScores. The Frog class represents the frog that is moving around the screen. When it is initialized, the location, size, and picture are set. This class extends ImageView in order for the frog to be represented by the image of a frog. It has static

attributes of starting X and Y positions. Any time the Frog must return to the starting position, the controller's call to the Frog's restart Frog method sends the Frog back to the original position in the Scene. Another important attribute of Frog is its flag variable that indicates whether or not it is currently in contact with a Water Object. This Flag helps the WaterObjectCollisionTask thread know whether it is catching an actual Frog-WaterObject collisions, or if the Frog is already riding a WaterObject to begin with. This flag variable helps avoid redundant checking and threading procedures. Another object that is represented using a class is a MovingObject, which also extends ImageView. These objects represent cars, logs, and turtles, depending on the image used. A MovingObject also sets its image, starting X and Y positions, and size when it is created. It also creates its Path and PathTransition, which are classes from the JavaFx Animation API. The Path and PathTransition are what gives the MovingObject its ability to move as well as its movement trajectory. Another object is a River. A River object acts like a list of MovingObjects that are more specifically turtles and logs. Similarly, there is a Road class. The Road object is like a list of MovingObjects, but more specifically, cars. This helps to set up the road that the frog has to cross. Shown below in a piece of the overall UML diagram is the relationship between the Road, River, and MovingObjects. The Road has an instance variable of a list of MovingObjects. This list will always be of length 5 because that is the number of cars there are for each road. Similarly, there is a list of 5 MovingObjects as an instance variable for the River because there are 5 logs or turtles per river.



On the screen, there are 5 River objects and 5 Road objects. This is why when the user is playing, there are 5 rows of cars and 5 rows of water objects (logs and turtles). This is shown below in the UML diagram because the FroggerView contains 5 Road objects and 5 River objects.



Also shown in the above diagram is that the FroggerView class contains 1 Frog object and 5 LilyPads, which is another object used. A LilyPad object also extends ImageView and initializes its location, size, and the image used. However, LilyPad objects never move. Instead, they are either occupied by a frog object (if a frog lands on it) or unoccupied (if there are no frog objects on it). Finally, there is the HighScores class. This class controls the sending and retrieving of high scores to and from the text file.

We tested our program thoroughly throughout the entire agile process of our product development using simple print statements as well as break points with the debugging console.

However, when it came time to implement the JUnit tests, we found that JUnit tests are not compatible with Javafx, as we get an “internal graphics not initialized” error. To work around this problem, we stuck with our regular print statement/ break point forms of testing as our way of actually implementing working tests. In addition, however, we did write the code for JUnit tests for the necessary methods of our project (to show that we did put in the effort to create them), we simply commented out the error prone tests to avoid the problem that was out of our control.