

Object oriented programming and design

Second semester project

Smart Piano

Engineering Department
La Salle – Universitat Ramon Llull
May 24, 2020

Name	Login
Amir Azzam	Amir.azzam
Felipe Perez	Felipe.perez
Arcadia Yaulten	Arcadia.yaulten
Sonia Leal	Sonia.leal
Nicole Leser	Nicole.leser

Contents

Introduction	3
GUI	4
Server	4
Main Menu.....	5
Manage Songs	6
Top 5 Songs	7
Plot Song Plays Evolution	8
Client	12
Login Block	13
Menu Block	14
Piano Block.....	17
Error Dialogs.....	22
Model Design	24
Class Diagram.....	24
Server	24
Client	29
Shared module.....	32
Description.....	34
Server	35
Client	40
Development Methodology	44
Time Costs.....	46
Analysis and Design.....	46
Coding	47
Documentation	47
Conclusion.....	49
Bibliography	50

Introduction

For the second semester Object Oriented Programming and Design project, our group chose to do the Smart Piano project. The purpose of this project was to develop an app that allowed users to play the piano and interact with other different users by using a Server-Client system architecture. We had to implement a server and client side, and their respective GUIs.

The purpose of the server side was to store access data, save the songs, and manage other communications. In the requirements, the server side had to register/authenticate users, manage public and private songs, graph the number of plays and the total minutes of music played, and show the top 5 most popular songs. To complete these objectives properly, the server side was split into two parts, a remotely accessible server, and a server interface. The remotely accessible server would be a MySQL database which could manage requests from clients by using TCP connections. This MySQL server would store all user data, all songs that users have created, and all statistics about the songs played. The UI on the other hand, would show all the songs on the server, the top 5 most played songs, and graph the number of plays and total minutes played.

The client program would be available to all users, and once their account was accessed, they could play songs by communicating with the server. The client side had a few more functionalities than the server. The client should be able to play the piano for fun, record a song, or play a song that someone else has recorded. Additionally, the client should be able to find and search for friends, logout or delete their account, see their profile, and register an account or login with a pre-existing one.

The language that was used to do this project was Java. This allowed us to program the interfaces with functions from the Java Swing package, and fully utilize the Java utils package. The language that we used for the database was MySQL, since it was required, and it is the database language we are most familiar with.

GUI

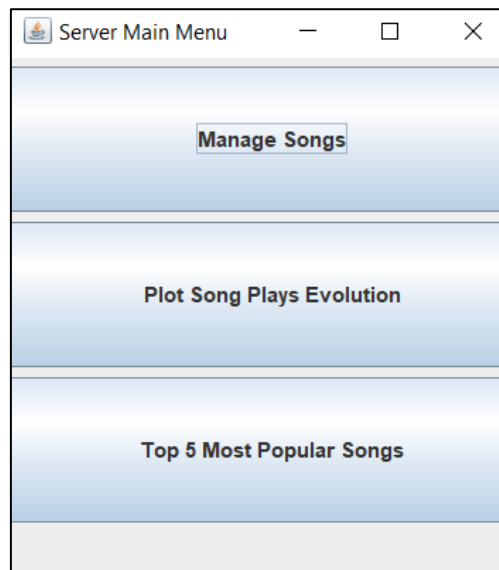
Server

The server GUI consists of 4 windows, the main menu, the manage songs table, the top 5 songs, and the graphs that display the statistics of the songs. Generally, the GUI is implemented as follows: Firstly, when the server is initialized, we create a JFrame. Whenever a user clicks on a button, a function is called to create a JPanel of the view that we desired. Once the panel is constructed, we place that panel in the JFrame, and display it to the user. Initially, once the server is run, a main view is created, and a main controller is created. The main view contains all the information that we need to show each of the views, and it contains one instance of each view we created. If any of these views contain a button, then we create the button here and save it as a class variable. This is to prevent problems when registering buttons in the controller, before the desired view is on the screen. Once the view is created, the main controller assigns all the listeners to the different views by registering the buttons.

Whenever a button is clicked that takes the user to a different view, the following process happens:

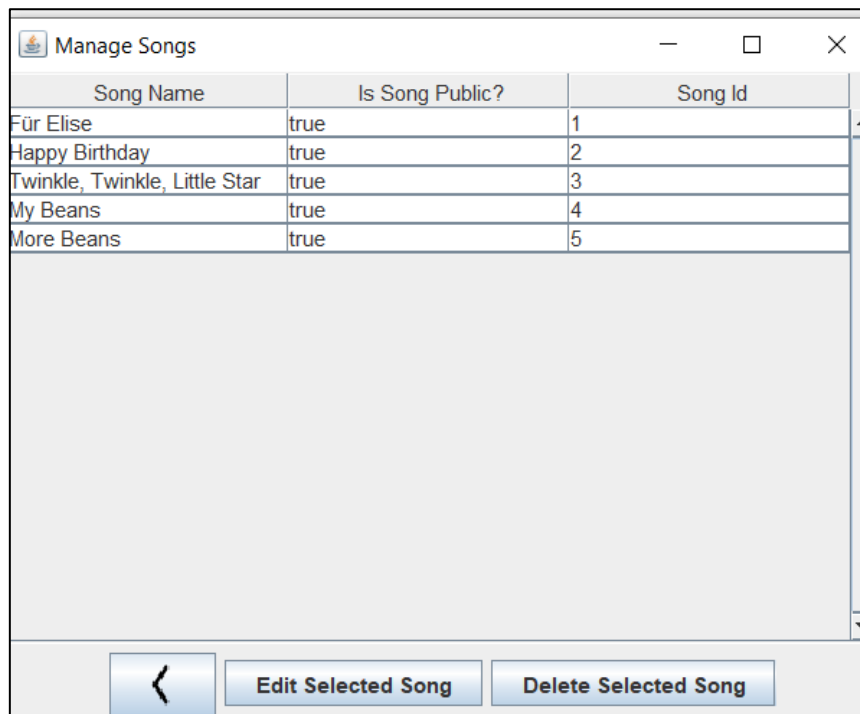
1. Because all the buttons have already been registered with action listeners and action commands, whenever a user clicks a button, the information about which action command it has is retrieved.
2. The function to go to the new view is called in the controller, which calls the same function, but this time in the view
3. The view we are going to is created by putting all of the appropriate elements onto one JPanel
4. From there, the previous JPanel is cleared, and the new one is added to the frame and repainted.

Main Menu



The first window that the user sees when they log in to the server is the Server Main Menu. As implied, the Manage Songs button leads to the manage songs view, Plot Song Plays Evolution leads to the graph view, and the Top 5 Most Popular Songs leads to the top 5 view. This GUI has a Box layout with center alignment. The three JButtons all have action listeners, and their action commands let the controller know which view each button needs to go to. This view is quite straightforward in function and implementation.

Manage Songs

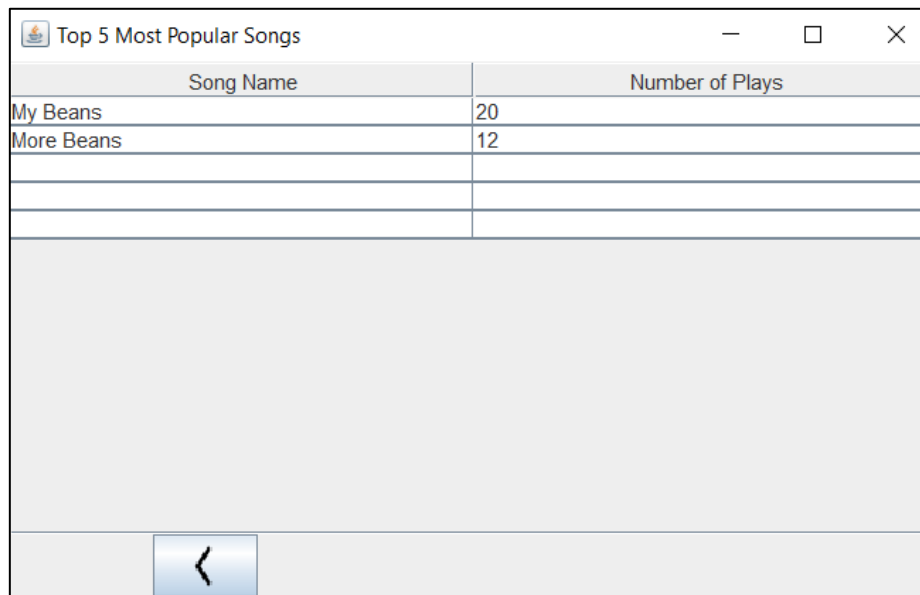


The manage song view is used to show all songs that are stored in the database, either public or private. Additionally, when a row is selected, the user can either choose to delete or edit a song by selecting a row and then clicking the corresponding button. Much like the main menu, the overall layout of the JPanel is a Box layout with center alignment. Within this box layout, there are 2 main parts: A flow layout at the bottom with the JButtons, and a Jscrollpane with a Jtable. Within the JTable is all the songs contained on the server, public or private, showing the song name, if the song is public or not, and the song's Id. If the user wants to change a song from public to private or vice versa, all they must do is click on the desired row on the table, and then click Edit selected song.

Something unique happens when the Manage Song View is created. Instead of just being a normal JTable, the song table is a DefaultTableModel, which is then added to a regular JTable. This is done to ensure that we can edit or change the information in the table with the click of a button. Once the JTable has been created, and all the server information added to it, the JTable is sent to the Manage Songs Controller via many different encapsulated functions. Much like the process earlier, the Manage Songs Controller determines what action has been taken by checking the action command. However, instead of changing the view, we check if a row has been selected. If a row has not been selected, then nothing happens. Otherwise, we check if the song is listed as public or private. On the click of the edit button, not only will the model change, but the corresponding information in the database will change.

On the other hand, if the user wants to delete a song, then they must select a row, and click the button. The deletion process is functionally the same as the edit process, however, at the last stage, the song is removed from the model, and the database as well.

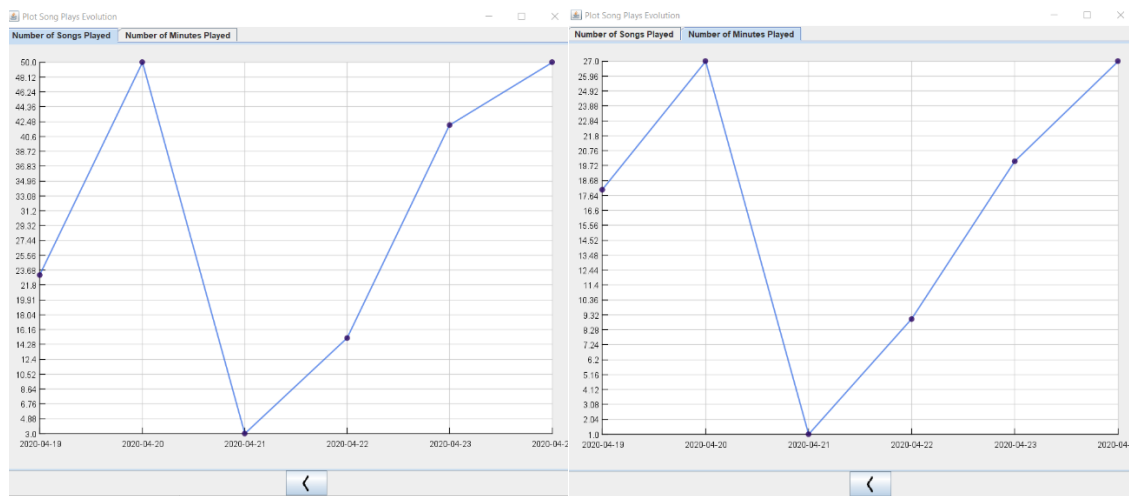
Top 5 Songs



Song Name	Number of Plays
My Beans	20
More Beans	12

The top 5 view is used to show the top 5 most played songs. The main panel for this view is a JPanel with a central aligned box layout. Within the main panel, there's 2 panels, an upper panel containing a Jtable, and a lower panel containing a single JButton. If a cell is selected in the upper panel, users are not able to edit the information inside. The information is displayed in descending order, with the first row being the song that's most played. Additionally, when a row is selected, the user can either choose to delete or edit a song by selecting a row and then clicking the corresponding button. The information in the table does not have to be organized, as there is a database call that gets the top 5 most played songs.

Plot Song Plays Evolution



The plot song plays view is used to show the graph of the total number of plays of all songs, and the total minutes played. The main panel for this view is a JTabbedPane with two panes, one with the heading “number of songs played” and “number of minutes played”. Both contain graphs created with the graphics library. The lower panel has a standard Flow Layout containing a single JButton. To create the graphs in the center, many different components had to be used to create the appropriate image.

Much like the other views, whenever we want to display a GraphView, we call a function that returns a JPanel with all of the appropriate information. In the graph view, we create the JTabbedPane, and then we create the two panels that we desire in the tabs. However, both of these panels are built in a unique way. There are two other view classes, GraphMinutesView, and GraphSongPlayView. These views are both used to construct the graphs that are seen on the tabbed pane. At face value, it seems that the only thing that happens in the function called is that the min and max values are retrieved from statistics, the class is set as the main panel, and returned. However, there is an important function called `paintComponent`. Whenever functions like ‘`repaint`’ or ‘`resize`’ are called, the function `paintComponent` is called internally, so it is never directly called by any specific function. So, although very little actually happens in the ‘`addPanels`’ function, the graph is correctly displayed whenever `paintComponent` is called.

The `paintComponent` functions has most of the code that makes the graphs display correctly. It should be noted that most of the functions in `paintComponent` are equations that use general constants for their functioning. For example, padding and `labelpadding` are used to determine how much size should be spaced from the edge of the window and start of the graph axes. `pointWidth`,

and numberYDivisions are also important constants, determining how thick each point should be, and how many values we should have on the Y axis respectively.

The first thing that needs to happen is initialization for using Java's graphics class. In order for paintComponent to work, we need to use the Graphics class from Java AWT. In our case, we pass it a Graphics2D object that we create whenever we want to paint the JPanel component. Once we have correctly initialized the graphics, we can then start doing calculations to draw our graph correctly. Firstly, we determine the size of the X and Y axes. The size of the X axis is dependent on how many items are in the statistics array, and the size of the Y axis depends on the Max and Minimum values determined earlier. The formulas to determine the sizing are:

$$xScale = \frac{window_width - (3 * padding)}{size\ of\ statistics\ array - 1}$$
$$yScale = \frac{window_height - (3 * padding)}{(maxElement - minElement)}$$

The numerator of the formula is used to determine the total numerical size of the x and y axes. Since the X-Scale is dependent on time, we divide it by the size of the statistics array, as each position represents a date. On the other hand, the Y axis represents either the total number of minutes, or the total number of song plays.

Next, we have to convert all of the statistic information to concrete graph points. Since the information in the arraylist is already ordered in chronological format, we determine the x position by it's position in the statistics array. Additionally, we take factors such as the scale of the X axis and the padding into account. We determine the Y position by subtracting the number of minutes from the MaxMinutes, so that we can appropriately offset each point on the graph. Here we also take into account padding and the Y axis scale. Then, using those coordinates, we create a new Point object, and add it to an arraylist of points.

After calculating all of the correct dimensions for the graph, now we can start to draw it. After setting the color we want our object to be, we create a rectangle. The rectangle uses the dimensions that encompass the entire window, bar the padding.

Since we wanted to have precision, we put grid marks on our graph. Since we want to be able to see the graph lines clearly, we made the color of the grid black. An arbitrary value is put for the number of divisions that we want. In our case, it's 25, however, it could take any other value (within reason). For each hatch mark on the y_axis, we determine the height by spacing by subtracting the

overall height from a ratio of the total height and the position it should be in, divided by the number of y divisions there should be. On the X_axis, we use the same techniques, however, instead we calculate where each line should go by multiplying the current index by a ratio of the window width to the size of the statistics array.

As we're making the grid marks, we also make the labels for each axis. For the Y axis, this was just a matter of putting a dynamically calculated value for the number of songs played, or the number of minutes played. We chose this method because we were unsure of what the upper limits on the graphs might be. Since we were unsure of how to generate this value ourselves, we did some research online and found the following formula:

$$\frac{(min_element + (max_element - min_element)) * ((\frac{i}{number_Y_divisions}) * 100)}{100}$$

For the X axis, we simply chose the date, as we had to graph the statistics against the time. We ended up using days as our units, as it would allow for a more comprehensive overview of all of our data.

Additionally, we had a problem when we had only one item in the graph as finding the X scale and the Y scale divides by size of the statistics -1 which means that we would be dividing by a zero, that's why we added a condition to avoid this kind of problems.

Then, after changing the color we want to black, we start to create the grid lines for both axes. Finally, we draw two lines for the axes. Since we are drawing a line, we need 4 points in total, The size of these axes are determined by x_1, y_1, x_2 , and y_2 . Where x_1 and y_1 are the starting coordinates, and x_2 and y_2 are the ending coordinates. In both cases, the value for x_1 and y_1 is the initial padding for the graph + the padding for the labels, and the height minus the padding and the label padding, respectively. This sets the initial point at the '0' point of the graph. On the y axis, the second x point is the padding and the label padding, as we want both points to have the same x points. The second y point is just the padding, which means that it is just off of the top point of the window, so it takes up an appropriate amount of space.

On the X axis, while the two y points are the same (height minus the padding and the label padding) the x values are different. The second x value is the width minus the padding. This means that we draw the line between the '0' point and just before the edge of the window, again, an appropriate size for the graph.

These axes mean that the graph takes up nearly the full width of the window with the exception of the label padding and the default padding we chose. Given that we are adding them to a tabbed pane, this is a good size.

Next, we draw the lines that connect each point on the graph. Firstly, we save the previous basic stroke that we were using on our graph, so that we can use it later for the dots. In Java, a Stroke is an interface that outlines a Shape enclosing a certain area. Stroke objects have different types of attributes, such as width, type of stroke, and many different features. In our case, we want the line width to be slightly thicker for the lines on the graph, we create a new BasicStroke with a size of 2, and use this stroke to draw the lines on the graph. No other attributes about the stroke are changed. Then, we loop through each point that we want on the graph to determine where we should draw the line. Firstly, we get the starting x and y position by getting the x and y value of the point that the loop is currently on. Then, we get the ending y position by getting the x and y value of the next statistic. Then, the line is drawn between the two points by using the drawLine function.

Finally, we draw each point. First, we set the color to a constant value that was pre-determined. Then, for each point, we have to adjust the x and y values for the points based on the point width. Since we manually determine point width and height, we do not use any stroke. Otherwise, the points could appear in slightly different positions than we want them to. To determine the correct position, we use the equation:

$$dot_x/y_pos = \frac{(x/y \text{ value of point}) - point \text{ width}}{2}$$

By subtracting the point width from the x/y value, we eliminate the offset which would otherwise occur. The division by 2 is used to ensure that the point is centered where we want on the graph. Since we want to draw a circle, the point width and height take the same value, so we simply use Java's FillOval function by passing it the previously calculated x,y,width, and height values.

One thing that should be noted is that in order to make the display the graph correctly, we return the instance of GraphMinutesView or GraphSongPlayView that we're working with directly. Normally, we would create the object itself outside the class, and run the functions on the object. However, when we were initially doing this part of the project, we had problems getting the 'paintComponent' function to properly run whenever we just tried to use a JPanel of the class type. However, by making both of the classes extend JPanel, and returning the class, we managed to get 'paintComponent' to work. If we were to redo this project, we would probably go back and make this implementation more sensible, however, it was created when we had very little knowledge of the 'paintComponent' function.

Client

The client GUI is the most robust GUI of the project. When navigating through it, though, you can see that it's very easy to understand. The view consists of three main blocks: the login, the menu, and the piano.

The login block is made up of two views, one to register the user, while the other one is to log in with a user. This view is used to collect user input and check the authenticity of the user.

The menu block consists of the menu and all related options you can select. The menu has JButtons, one for every functionality you can do in the application. When you click on any of these buttons it takes you to the corresponding view, there is an option to log out and another to delete account both takes you back to the login block. And there are options to play the piano which will move you to then Piano block.

The Piano block consists of the piano itself and all related functionalities, like playing the notes, reproducing the song, ...etc

The way we connected all the views with the rest of the project is using the Main View class.

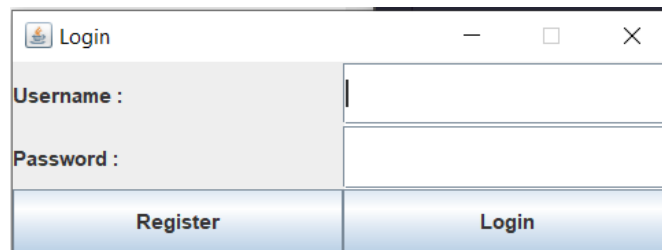
The main view, also, holds the main JFrame for the client-side, in some way it acts as a central manager that organizes what is being displayed on the screen with the help of the controllers. This main view contains an instance of the other views and contains the functions that refresh the JFrame whenever the user wants to move to a new screen. Whenever a "moveToX" function is called, there is a function of that specific view that creates the specific JPanel needed. After it's created, the title is set to something more appropriate, and the correct dimensions of the frame are registered. Then, the panel is inserted to the JFrame, and the repaint and revalidate functions are used to display the panels correctly.

Additionally, the main view is the only class between the client view classes that communicates with classes from outside the view package. That being said, it is the one in charge of providing the required information to the views that need it, and also is the one in charge passing the user input to the controllers if the interaction is not monitored by any listeners.

Note: the way this class communicates with the rest of the system is explained more in the [Client](#) diagram section

Login Block

Login View



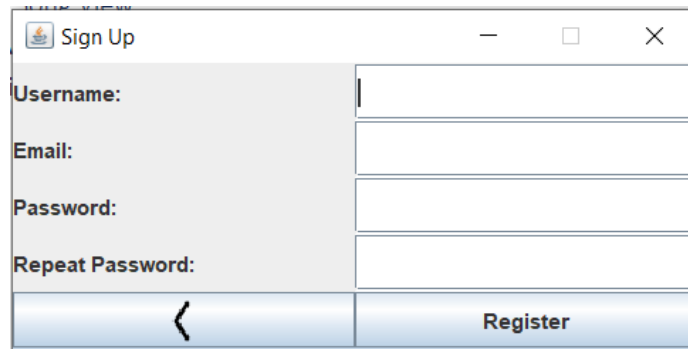
Username :	<input type="text"/>
Password :	<input type="password"/>
Register	Login

The login view is the first view that users see when running the client application. It is a JPanel that we fill with the information to be displayed. The layout is a GridLayout, split into 2 columns and 3 rows. Firstly, we create the 2 labels (Username and Password) and set the text. Then, we create the 1 JTextField for the inputting the username and a JPasswordField for inputting the password. Once created we add them to the panel in order, so that they can be displayed properly. The order is: Username label, username text field, password label, password field, register button, login button.

These two views have two buttons each. The login has a button to submit the credentials and login if they are correct, moving the user to the menu, and the second button is used to move the user to the signup view in case they are new in the system. Similarly, the signup view has one button to submit the credentials and register the user, and another button to go back to the login view. Moreover, these views have input boxes to collect the needed information from the user, we have a method called getInput() that calls the getText() method on the field objects.

The system checks if the username and password are registered on the system. If they are, then the user is redirected to the Menu View. If the user enters any incorrect information, then a dialog box pops up, indicating the source of the error.

Register User View

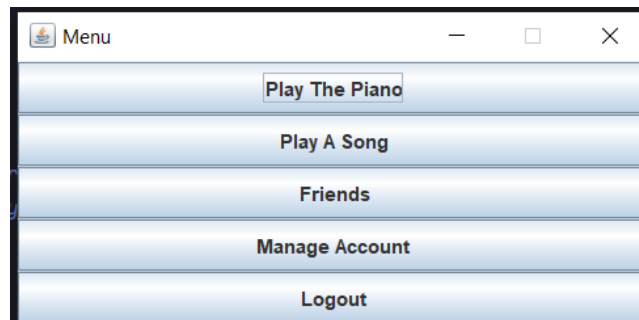
A Java Swing dialog box titled "Sign Up" with a standard title bar (minimize, maximize, close buttons). The dialog contains a light gray panel on the left with labels for "Username:", "Email:", "Password:", and "Repeat Password:". To the right of these labels are four white text input fields. At the bottom of the dialog, there is a blue button with a white left-pointing arrow and another blue button labeled "Register".

The register user view is the view the users are taken to whenever they click Register on the login view. The layout is a Grid Layout 5 rows and a single column, with each column containing one of four JTextFields (two of which are JPasswordFields) and two JButtons.

If there are no errors when the user clicks the register button, the user is taken directly to the Menu view. Much like the previous sections, to collect the needed information from the user, we have a method called `getInput()` that calls the `getText()` method on the field objects. If the user inputs any information that is incorrect, then an error dialog box appears, informing them of their error.

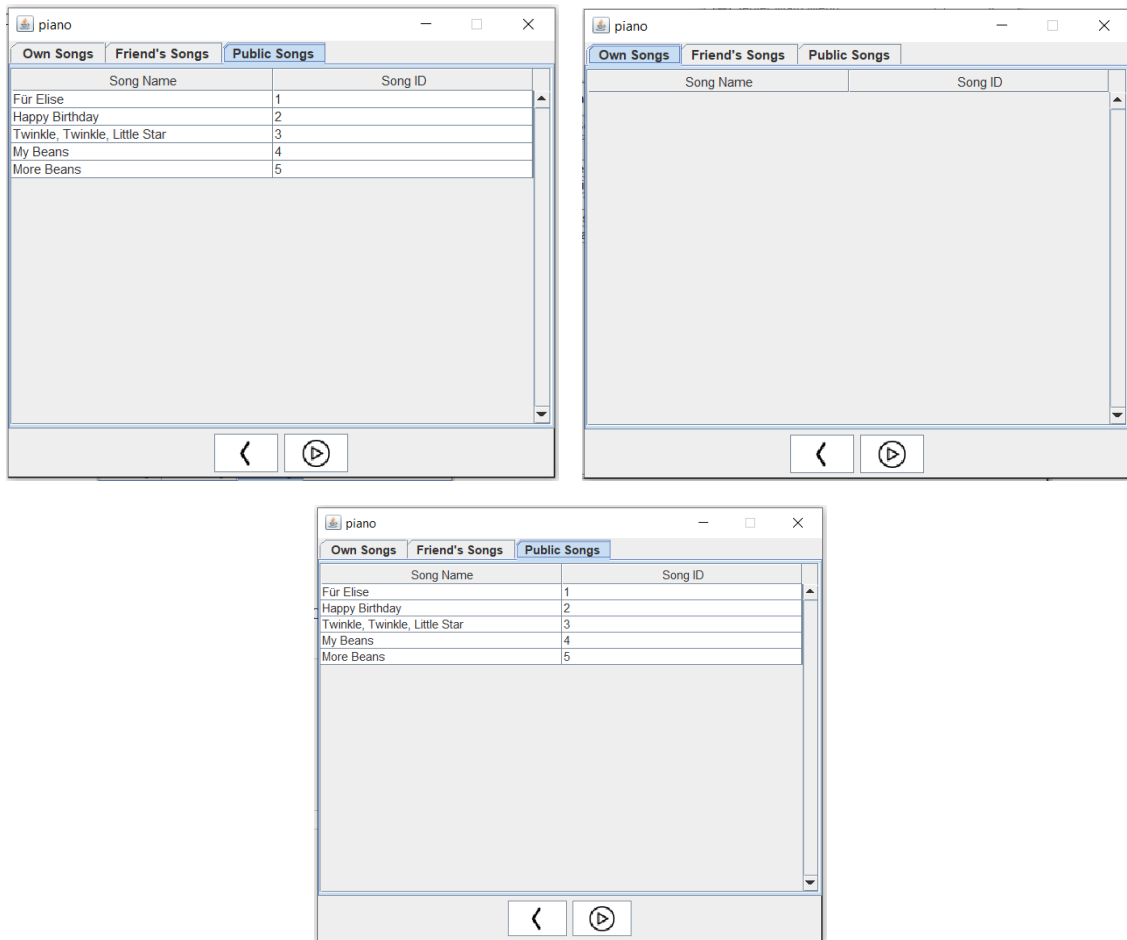
Menu Block

Menu View



The menu view is a vertical Box Layout JPanel consisting of 5 JButtons, each of which takes the user to a new view. Play The Piano takes the users to the Play the Piano View, Play a Song brings the users to Choose Song View, Friends takes the user to the Friend View, Manage Account takes the user to Manage Account View. The logout button logs the user out of the system. The button navigation has already been explained in the Server section.

Choose Song View

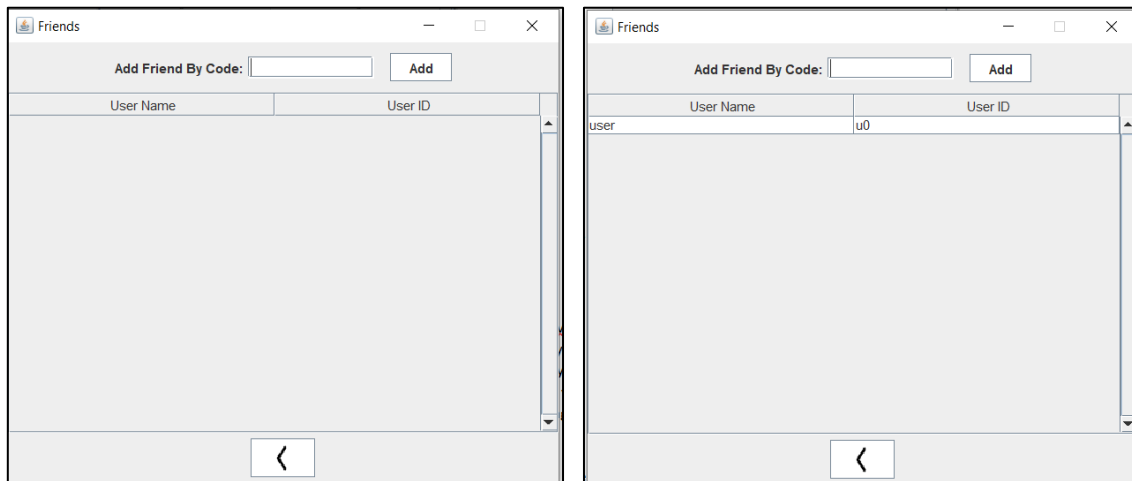


Shown in the images above are the contents of the different tabs of the ChooseSongView. This view allows users to directly access and play either their songs, their friends' songs, or any public songs. To play a song, all the user has to do is select the appropriate row, and then click the play button seen on the bottom of the view. They are then automatically taken to the piano song view. This view consists of 2 main parts, A JTabbed Pane and a JPanel with a Flow layout. The flow layout along the bottom contains two buttons, a back button, and a play button.

The JTabbedPanel contains 3 tabs, each with a unique JTable. The first JTable contains all the user's songs, the second, their friend's songs, and the final one, all the public songs. The information for these tables is retrieved from the database, and its information is simply filled in on the table. Additionally, if a user tries to double click on any of the fields of the tables, they are not allowed to.

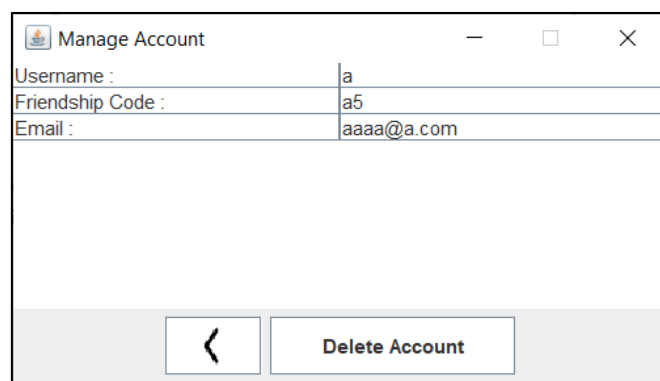
The back button functions as mentioned previously, however, the play button opens the appropriate JFrame and allows the user to play the song that they want by displaying the Piano Songs View.

Friend View



This is the view displayed when the user clicks the “Friends” button on the main menu. The JTable in the center is for displaying all of the friends that the user has, both the user name and the friend code. The JTextBar at the top allows users to input known friend codes, and then add them by clicking the corresponding JButton. The bottom part consists of a JTable, that is updated accordingly with each friend addition. To be able to perform the update operations, a DefaultTableModel is used, and all the info up to this point (all the friends a user has before they enter this view) is loaded and then a table is created by passing the DefaultTableModel object to it as a parameter.

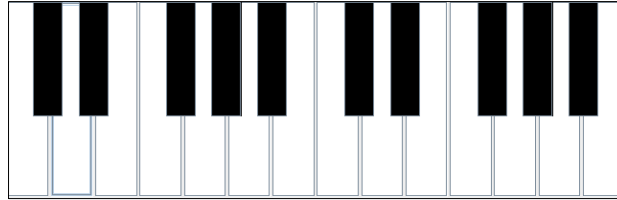
Manage Account View



Manage Account View contains two main parts, A JTable with the user’s information, and a JPanel with a flow layout along the bottom containing two buttons. The JTable contains all the user information: Username, Friendship Code, and their email. We know this information, because whenever this view is created, a user is passed to it. This way, we can just run the getters registered to the User object and get the information that we need. The Back button simply takes the user back to the main menu. However, the delete account button deletes the user from the database.

Piano Block

Piano Keys

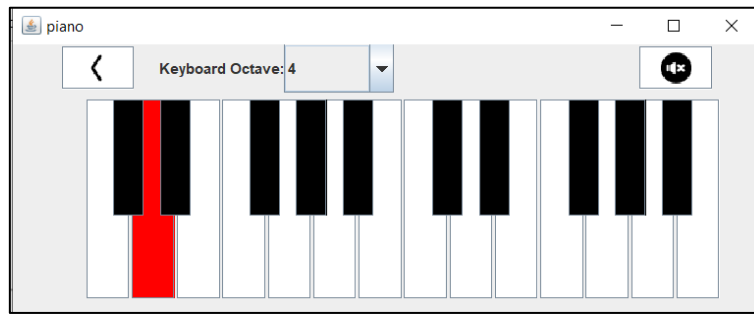


The piano keys view is a `JLayeredPane` we use two different layers to create the view pictured above, the first layer holds the white keys, while the second layer holds the black keys.

For this view, we require an action listener, a mouse listener, and a keyboard listener. The process in which we create the buttons starts by creating a `JButton`, then we assign the note that button will hold, and we position it in the `JPanel`. Lastly we assign it an action command representing the note it plays, and we assign it the appropriate listeners. The process is done in two loops to create all the buttons. The first loop creates all the white buttons, while the other creates the black buttons. The difference between the two loops are: the layer in which we position the button, the dimensions of the buttons, and finally the color of the buttons. In both cases, we save the `JButton` we created with its name in a `HashMap`. We can use the `HashMap` to change the color of the `JButton` to indicate that it has been clicked when reproducing a song.

This panel is created only once and a reference to the `JPanel` is saved so we don't have to recreate it every time we need to display the piano keys. Instead, we return the reference to the already created panel.

Piano Songs View



The piano-songs view is displayed whenever a user wants to play a song that has already previously been recorded. Unlike all the other views, this view opens as a pop-up, and to do that we create a new JFrame in the main view, in which we put the panel pictured above. This frame uses the JPanel created by the Piano Key view to display the keyboard of the piano and use its related functionalities. This is possible as the listeners to the keyboard are already established when the JPanel is created.

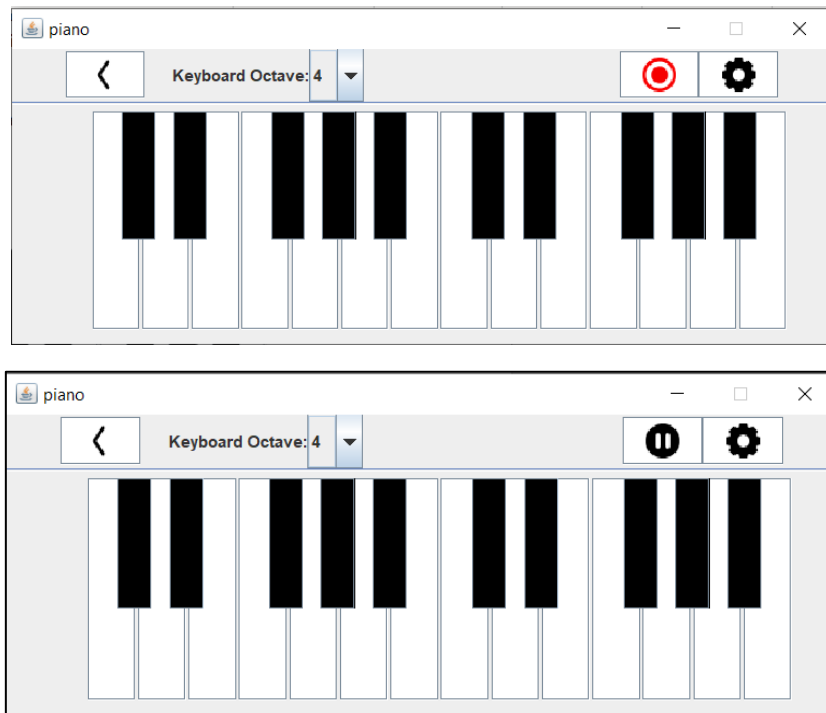
This Panel you can see in the picture above consists of a BorderLayout, the piano keys panel is added to the south, in the center we have a JSeparator to separate the north and the south.

In the north we have another horizontal BoxLayout, to that, we add two JButtons, one is the back button and the other one is the mute button, and we have a JLabel that says “Keyboard Octave: ” then next to it we have a JComboBox to choose the octave you want the keyboard to play the notes on.

Once a song is selected from the menu and the play button is clicked this view will pop-up in the center of the screen, the keys will highlight red, indicating which specific note is played. At any time, the user can change the octave by clicking the JComboBox, or mute the song by clicking the JButton with the mute symbol. The user can also return to the previous screen by clicking the JButton with the Back symbol on it. Once the user is finished playing the song, the window closes automatically.

The action listeners to the JButtons on the north are assigned via the main view after the instance of this class is created.

Play Piano View

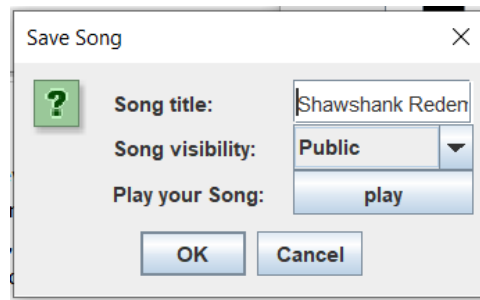


The Play Piano view is displayed on the screen when users click the 'Play the Piano' button from the main menu.

This view is very similar to the one before, it also has a BorderLayout with the piano keys inserted to the south. The center has a JSeparator separating the north and the south. The only difference between this view and the one before is the north.

In the north, we still have the JLabel and the JComboBox from before. However, we have three JButtons instead of two, the one on the left is used to go to the previous screen, The JButton with the gear icon leads to Keyboard Settings View. Finally, we have the record/pause button. This one JButton but we change the image inserted to it. The minute you click this button it changes into the pause image and the program starts recording every click you do on the piano keys (whether it is using the keyboard or the mouse). And once you are done you can click on the pause icon, and that will stop the recording and change the icon back to the record icon. Once the process of recording one song is done this will show a pop-up to finish up the recording of the song and save it.

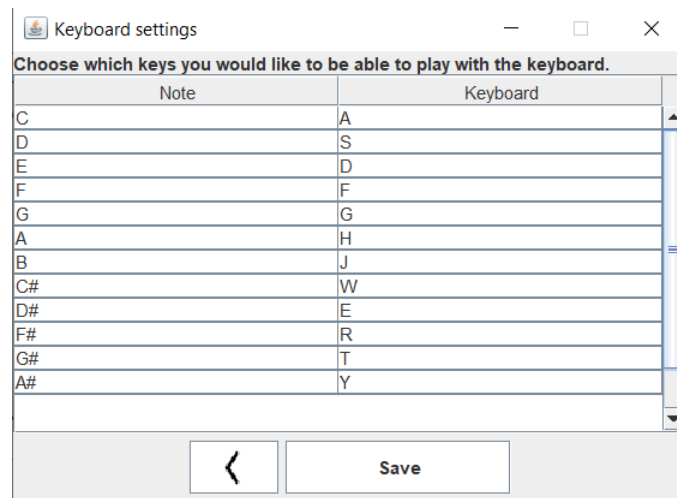
Save Song Dialog



When the user has finished recording a song, and they click on the stop button on the Play Piano View, this dialog box pops up. This DialogBox contains many items, most notably, the JTextField, JComboBox, and three JButtons. This dialog box is somewhat more complicated than the normal dialog box. The layout is a GridLayout with 2 columns and 3 rows at its heart, but then we call the showConfirmDialog method of the class JOptionPane, passing it as parameters, null as the parent component (no parent), the frame we construct in the method (explained later), the title and indicate we want “OK” and “CANCEL” buttons (with JOptionPane.*OK_CANCEL_OPTION*). The text field is created the same way as in other views like this one, such as the LoginView. The Drop down box is created with a JComboBox, and passed a String array.

The OK button registers the song in the database, and the cancel button just closes the dialog box, taking the user back to the Play Piano View.

Keyboard Settings View



The Keyboard Settings view can be seen when the user clicks the gear icon displayed in the Play the Piano view. This allows users to change the keyboard mapping of the keys corresponding to the piano keys. These configurations are only saved until the user closes the program.

This view consists of a JTable with two columns added to the center of a BorderLayout, the north of that layout contains a JLabel that says "Choose which keys", and the south has a horizontal FlowLayout with two JButtons in it.

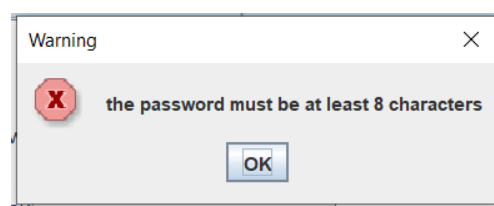
The table has only the second column set to editable while the first one is not. When a user double-clicks on the "Keyboard" column they must first erase the previous character and replace it with another one then unselect the cell so the table would know that the user is done editing, as no changes will be made if the user saves while editing. After the user is done editing and they click on save, then they would be able to play using the alternated keys. If the user enters more than one character, only the first one is taken.

Error Dialogs

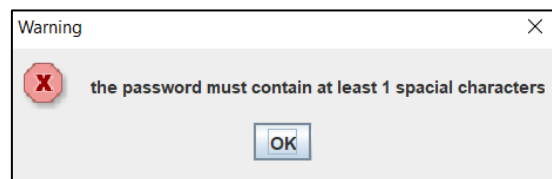
We have a method in the main view that given a string it generates an error dialog showing what is on the string. This method is used by the main controller to display the errors the user might encounter while interacting with the system.

These dialogs are warning dialogs so they have only one option to click which is a button that says "OK" clicking on it just returns the user to the view they were in and makes the dialog disappear.

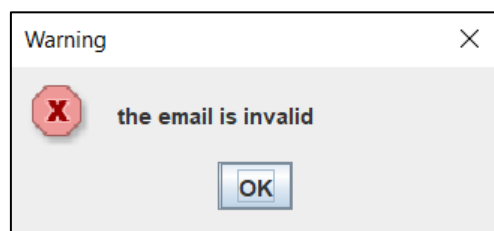
Here are some examples of the dialogs that can be shown:



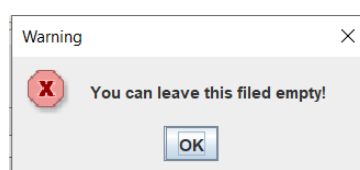
If the user inputs no information into the password field when registering a new user, this is the default dialog box that pops up.



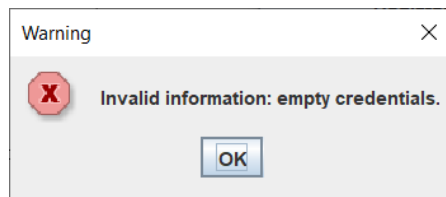
If the password the user entered when registering does not have a special character, this dialog box pops up



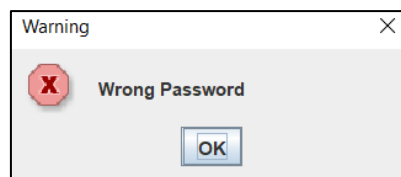
If the email address the User entered when registering is wrong formatted, then this dialog box pops up.



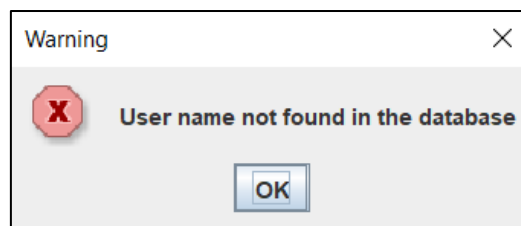
In the Friend View, if the user attempts to click on the 'add' button without inputting any information, this error popup is shown.



If a user tries to log in without entering any information on the login screen, the following message is shown. This error can be seen on the Login User View.



If a user tries to log in, and they enter the correct username, but a wrong password, the following message is shown. This error can be seen on the Login User View.

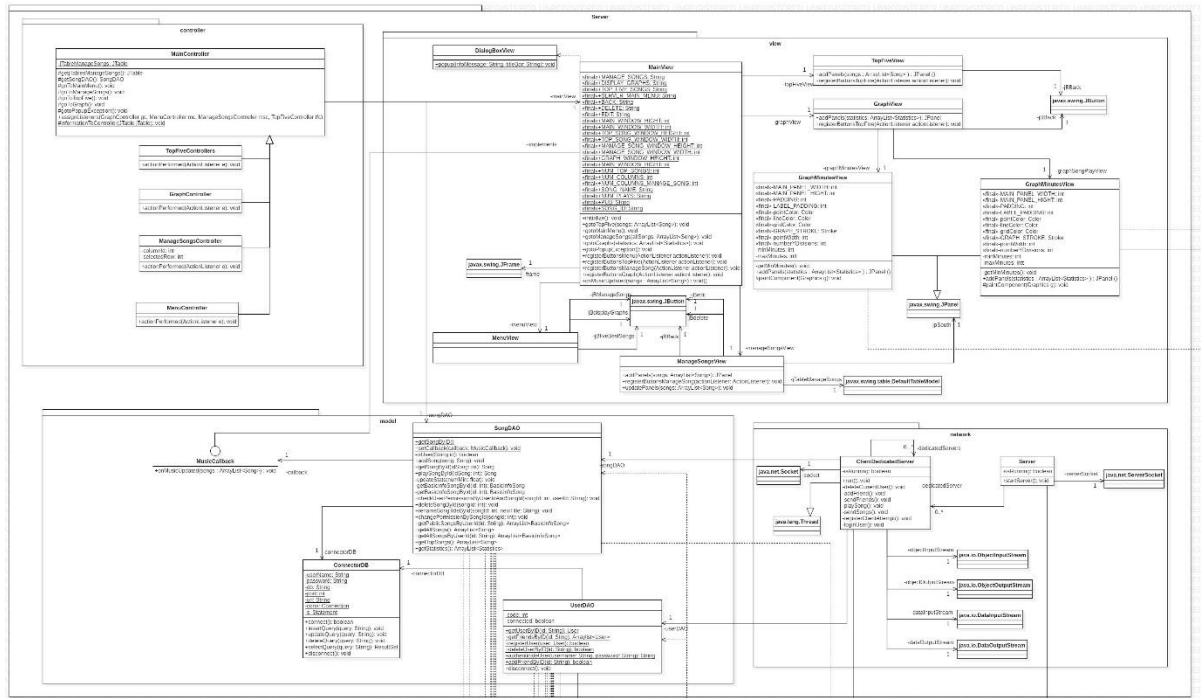


If a user tries to log in with an incorrect username, and a correct or incorrect password, the following message is shown. This error can be seen on the Login User View.

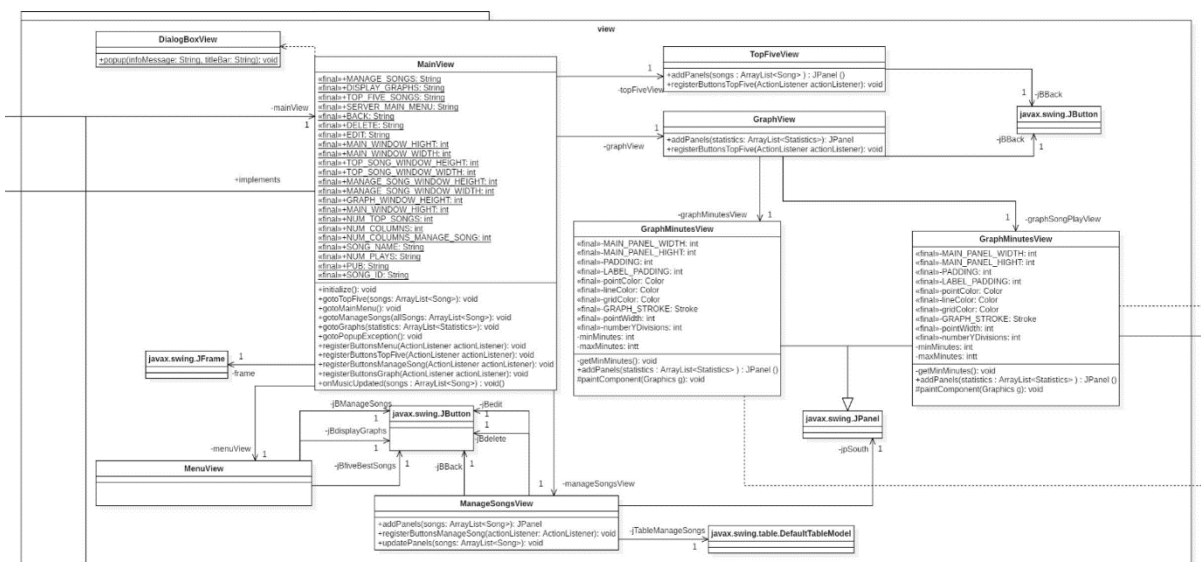
Model Design

Class Diagram

Server

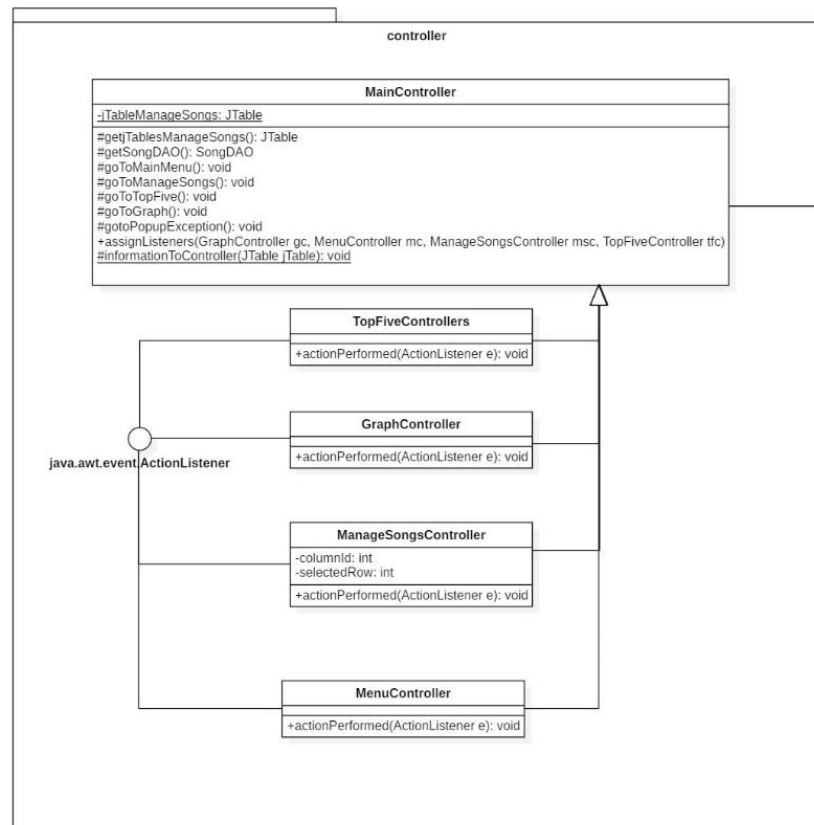


This is the class diagram for the server module, this module consists mainly of four different packages, module, view, controller, and network. These four packages are interconnected and communicate with each other for the server to function smoothly.



Looking at the view package you can notice how the MainView is the central class of this package, as it's the one that has the main-frame. The other view classes generate the panels and then the MainView takes these panels and inserts them into the frame it has, doing

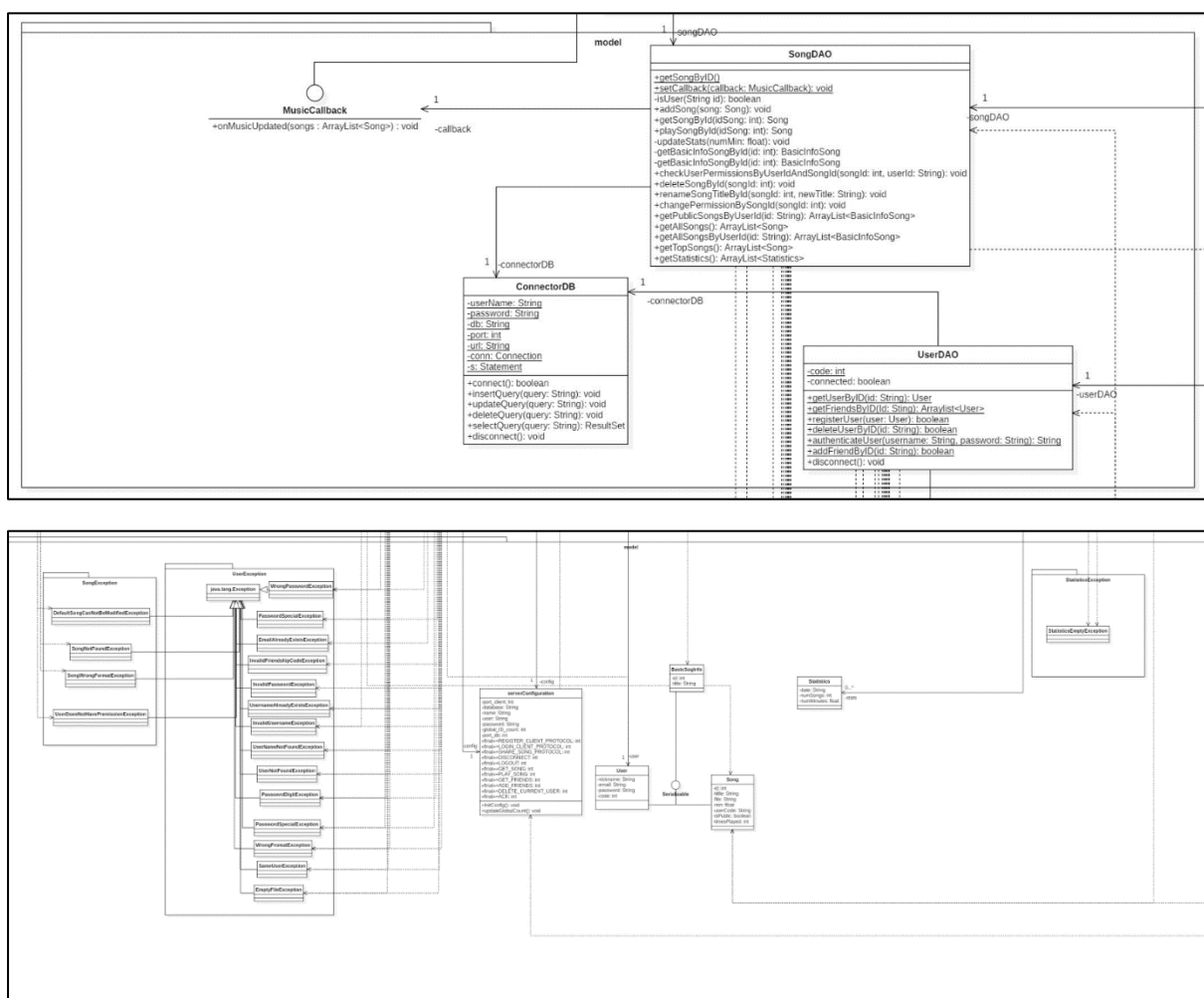
that we can prevent generating multiple frames each time we go into another view, we know that the frame can be destroyed, but we thought that this will be a cleaner solution as we can be sure that there are no two frames active at the same time since we only have one. The MainView is also responsible for assigning the action listeners for the buttons in the subviews as it has a reference to each of the classes. To understand better the design decisions we made over here, we need to look at the controller:



As you can see here, just like in the view, the main controller is the central node that connects all the other controllers, the idea here is that the main controller is the one that will communicate with the view and the model, then the sub-controller will be assigned to the components of the different views as action listeners, and if they need to access anything from the model or communicate with another view they can do that through the main controller. To be able to achieve that we made all controllers inherit from the main controller, and then we created some kind of shared methods between them by making the access type “protected”, so then all the children controllers can use these methods to communicate with the other packages. Furthermore, as the main controller has an instance of the view, then it creates a method in it to call each of the view methods as we didn’t want to make the access to the view attribute possible to all the controller children as we thought that the children

don't have to know all the information about the view but they just need the methods that are required to do the task.

This makes some sort of hierarchy, so when we first create the main controller we tell it which one of its children should be assigned to what view, then the main controller will tell the view to pass this information to each of its little views, then the main controller will be the one telling the view which view to display, then the view will get the panel from the corresponding view and display it, after that, if any of the children controllers need anything it will request the main controller to get the info for it or to do the action if it involves changing the view or accessing the model. Moving on to the model:



Here as you can see our model is split into two packages, and that is because the second package is located in the shared module, so it would be accessible from both the client and the server sides, the server's model has two main important classes, the UserDAO and the SongDAO which are the two database access objects in our project, using these two classes you would be able to access, change, and add information to the database.

There is a class called `connectorDB`, this class is responsible for executing all the queries done by the DAOs, that's why each of the DAOs has an instance of it. The `UserDAO` has a dependency on the `User` class but it might be not clear given all the dependencies that class has, due to all the exceptions it returns.

The same is for the `SongDAO` where it has a dependency on the `Songs` class. Both the `Song` and the `User` classes are `Serializable` as they are the classes that get sent back and forth between the client and the server.

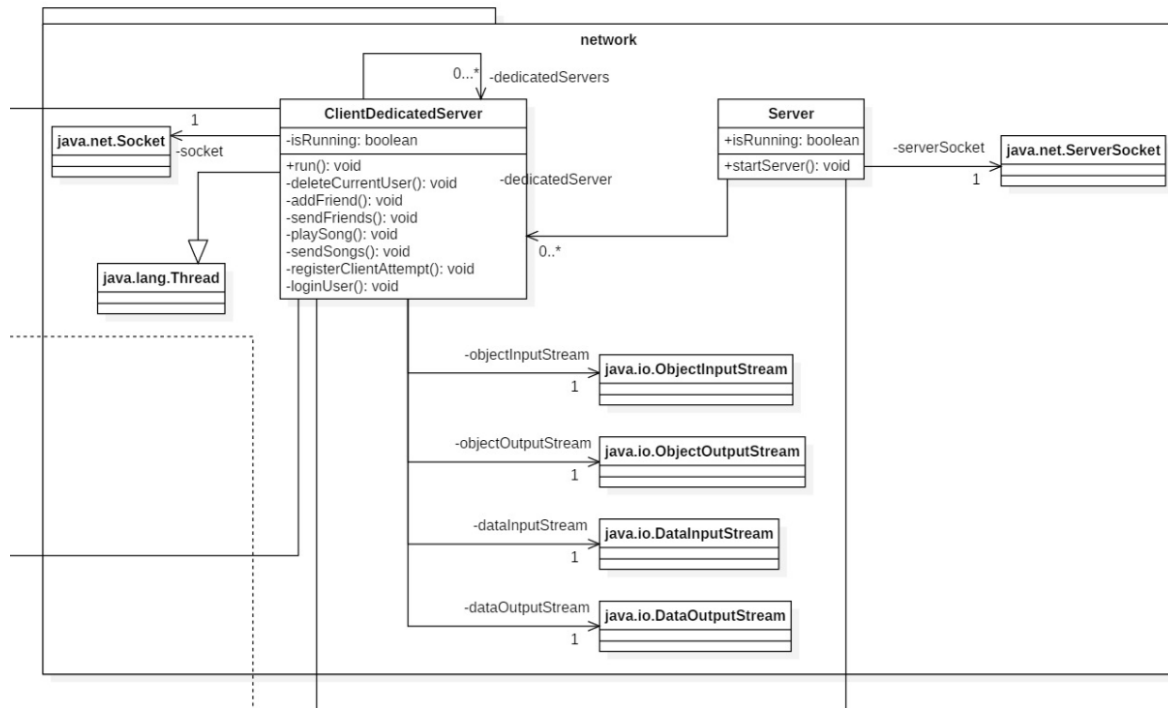
We do have another class that is serializable which is the `BasicSongInfo`, this class holds the basic information of the song without having any detailed information about the song nor the song content, and that is because we thought it would be more convenient to only send a list of the basic information of the songs rather than sending the full song, to optimize memory usage and control the client interaction with the database.

The `MusicCallback` interface you can see on the server's model is used as an observer, so when any changes on the songs database happen, the view will be updated and the new songs will be added.

The exception packages on the right, are all the exceptions we created and they are all thrown by either the `SongDAO` and the `UserDAO` or both, that's why they have dependencies on these classes.

Also, we have the class `statistics` which is used when returning the statistics from the `SongDAO`, it's not serializable as it's only used from the server-side, it was added to the shared module at first then we realized it can move to the server module and that would be the better option as it's more encapsulated.

Finally, we have the server configuration class that stores all the necessary information to connect to the database or to connect to the server in the case of the client. It also stores some information needed to initialize the system to the state where we left off.

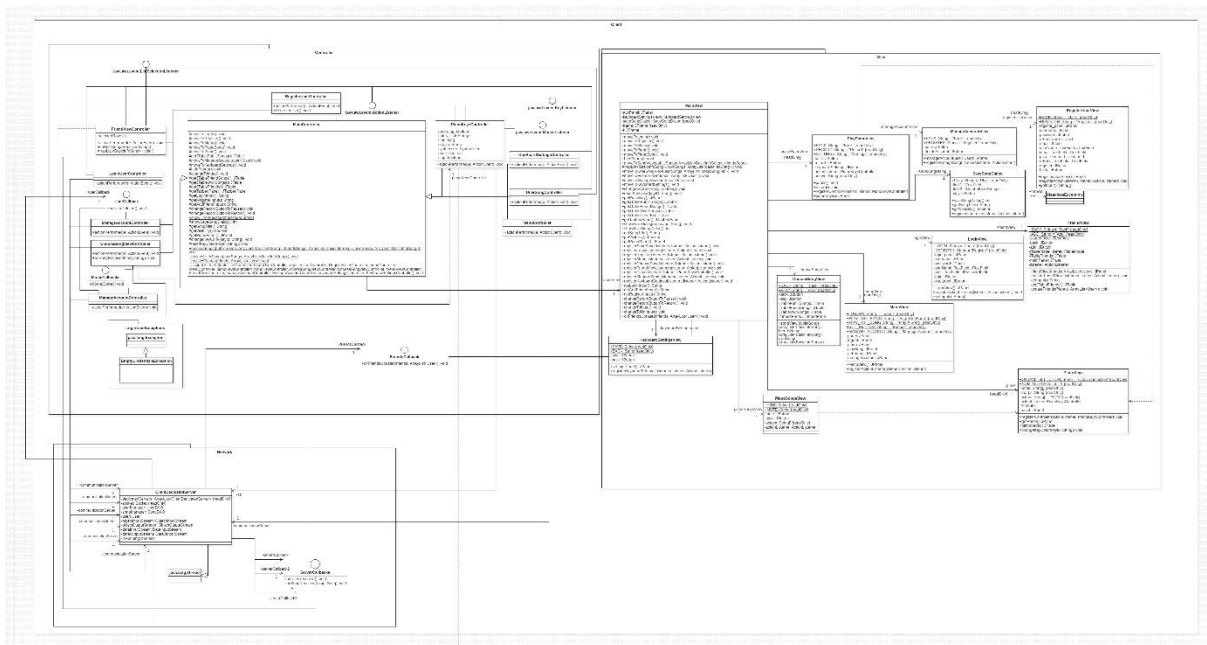


This is the network class from the server-side, as you can see in here the server has a dedicated server for each of the clients so it can answer multiple requests simultaneously without blocking the channels, the server class receives the connection request from the client, then it creates a dedicated server for that connection and a special socket is assigned to this conversation the whole time the client is connected. Each dedicated server knows the other dedicated servers, so when disconnected the reference to this list (that is shared between the server class and all the dedicated servers) will be modified, and the dedicated server will delete itself from the list, which will result in deleting it from the original list the server class has.

Each of the dedicated servers has an object input/output streams and a Data input/output streams, we use each depending on the type of data we want to send or the type of data we are expecting to receive.

Client

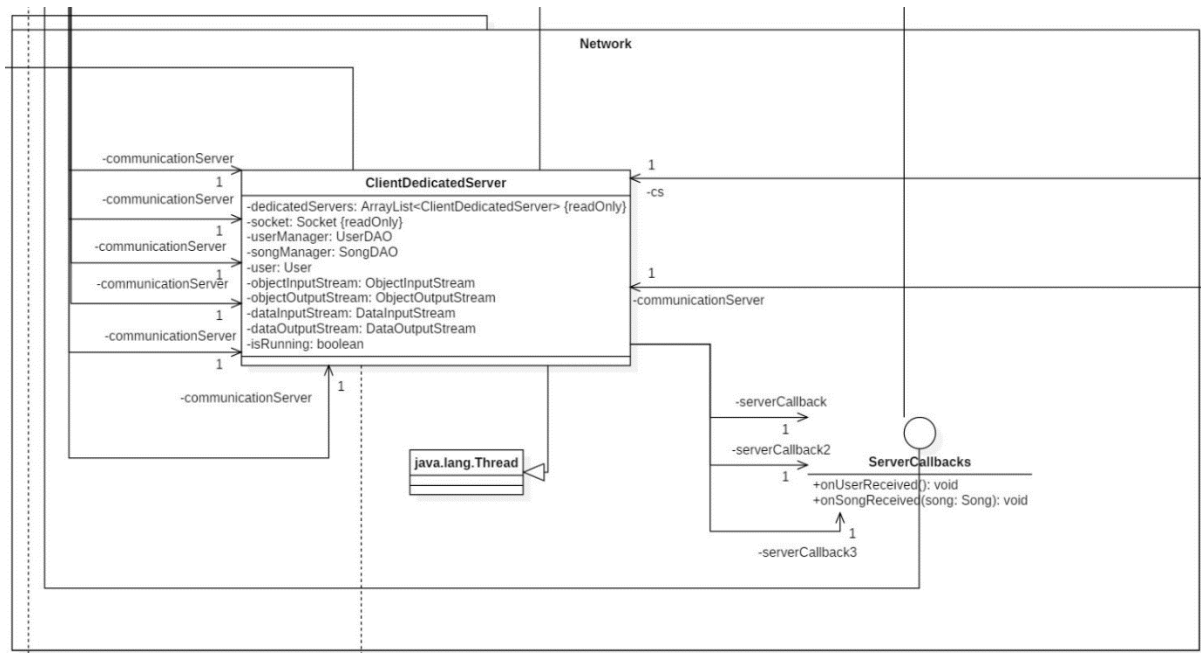
The client's structure is very similar to the server's with some changes when it comes to communication. This is the client's diagram:



As you can notice from the first look the client doesn't have any model by itself where it only uses the shared module package we talked about earlier, as it doesn't need to store information for itself, or at least we didn't feel the need to create a module to store the data we have on the client. That is because most of the data changes a lot and it's stored temporarily in most of the cases, and we re-request the data every time we move to a view, which is not the most optimal decision when it comes to performance, but it is an easy way to achieve the requirement. However, it has the three other packages from the server, which are the view, the controller, and the communication package (the network package). If we take a closer look at the view:

Finally, the network package is the one responsible for all the communications with the server.

This is the diagram of the package



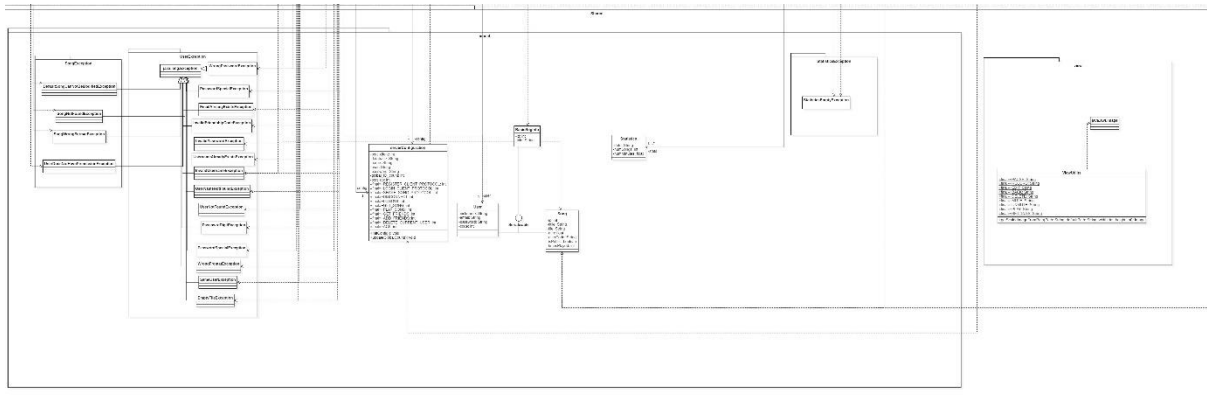
It consists of one class and one interface, the client dedicated server or the communication server is the class that connects to the server socket and sends and receives messages, this class sends requests to the server. A request can be a request for information or a request to change the database, either way, the server processes the information and returns the results to the client (which will be received by this class), then this class notifies the main controller using the servers callback interface you can see in this package or directly calling static methods on the main controller (we know it's not the best implementation and we should have done all of them using the callback, but we didn't have the callback at the beginning).

This class is instantiated in all the controllers that need to communicate with the server, and it is passed to the controller when creating it, the requests are then done directly from the controller.

Shared module

The shared module as we mentioned earlier, is accessible from the client and the server.

This is the diagram of the shared module:

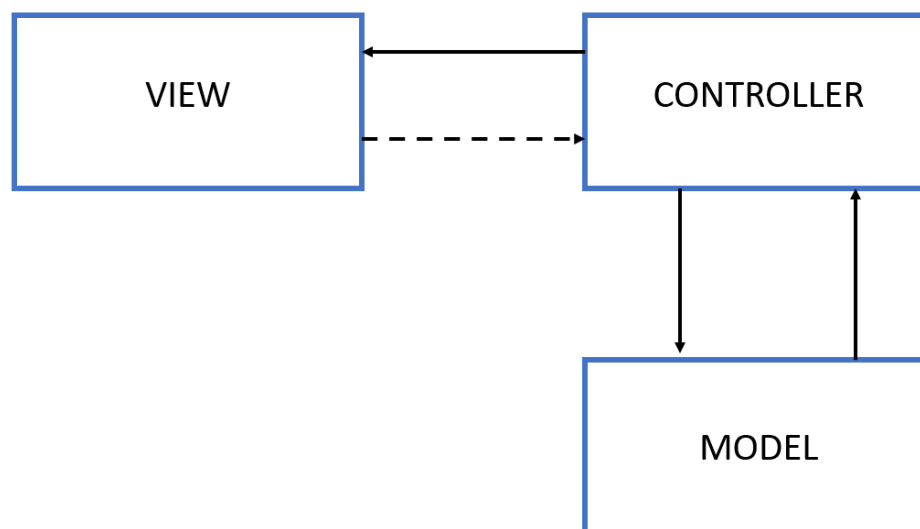


We have seen the model of the shared diagram before when discussing the server's model. However, we have another package which is the view, this package contains one class, this class's job is to load pictures into and returns an image object with the specified size. All the methods and attribute in this class are static, which was an attempt to create a single tone, we then learned how to create a proper single tone by having a static instance of the class in the class then returning it when access to one method or attribute is needed.

Description

The design pattern we have applied in the project is the model-view-controller. By using this design pattern, we were able to separate each functionality in distinctive classes. The model contains the data that will be handled by the controller. The controller and model both communicate with each other. If information must be obtained, the controller will notify the model of which information it must retrieve. Additionally, the controller can also require the model to update information. The controller is one of the most essential components of this design pattern as it is in charge to communicate with the view and the model. The controller acts appropriately according to the events that the view triggers. Depending on which event was triggered or how the user interacted, it will have to communicate with the model to obtain or change information and it might also have to notify the view of new changes. The last component of this design pattern is the view. The view represents the information graphically to the user that the controller provides and can also provide information to the controller if changes occurred.

The figure below illustrates the communications between these components:



Server

On the server-side of this project, we decided to create four different packages that correspond to each of the three elements of the model-view-controller design pattern and additionally another package that is specifically used for server functionalities. In each of these packages there are multiple classes for the many functionalities the program offers, but in this section only the more significant classes will be discussed.

The following are the most important classes in the server-side according to each package:

Server

- ClientDedicatedServer

The Client Dedicated Server class is used to assign each of the users one dedicated server which allows them to do functionalities required for the system to operate as it is intended. Moreover, multiple users can be logged into the system at the same time and still be able to use all the services the program provides because it runs each of the clients in different threads. When a dedicated server is running, it is in charge of communicating with the database to perform different data operations which are: attempting to register a new user, login in a user, saving a new song, disconnecting the client dedicated server, logging out a user, obtaining songs, playing songs, and many other things. This class administers the information that the client needs by contacting the database. In order to carry out the aforementioned operations, this class uses instances of these classes: `DataInputStream`, `DataOutputStream`, `ObjectInputStream` and `ObjectOutputStream`. The primary function for the Output Stream classes is to serialize data or objects (Serialization is a mechanism for converting the state of an object into a byte stream), and the other two classes to read/deserialize (do the opposite) information (`DataInputStream` for Java primitives while `ObjectInputStream` for objects). The `dataInputStream` received, an integer, determines which operations must be carried out. If something must be sent, first an integer is sent with the `dataOutputStream` to indicate the protocol and then the object can be sent using the `objectOutputStream`. The process is similar when information is being received. The main difference is that instead of working with `dataOutputStream` and `objectOutputStream`, it works with `dataInputStream` and `objectInputStream`.

- MainServer

The MainServer class, as its name indicates, runs the main program from the server-side. It only has one method which is crucial to make the system work for more than one user. The *startServer()* method starts the server and listens for new client connections. When a client wants to connect, it assigns a socket to this specific client. Moreover, it assigns a dedicated server through an instance of the *ClientDedicatedServer* class. Afterward, the server that has been assigned is started. Since many users can use the program at the same time, this class contains an array list that holds all the dedicated servers being used.

Model

- ConnectorDB

The aim of this class is to have access to the database (which is the localhost) to perform all the necessary operations that are included in the DAO classes. This class is one of the most crucial classes of the whole project as it needs to be able to provide the server access to the database in order to obtain and update information. This class has a JDBC connection that allows to perform queries to the database. It has methods that allow to insert, update and select queries. Before being able to use any of these query statements, the connection to the database must be satisfactory. The database language we have chosen to program the database with is MySQL. Thus, in order to be able to connect from Java we had to use a JDBC driver from MySQL. It is important to note that the information to connect with the correct password, username, and other information like the ports must be changed in this class if needed.

- DAO design pattern

The DAO (Design Access Object) design pattern is implemented to separate the logical aspect so other classes that require their methods do not need to know of how the methods are implemented which provides more security in terms of encapsulation. The classes with DAO provide methods to retrieve, update, and insert information without knowing its implementation. Although we have not implemented this design pattern like

many sources indicate with interfaces and other utilities, we have applied it to the best of our ability to assure the separation of logic principle.

- SongDAO

In this class, all the methods to obtain and change information in the *Songs* table and the statistics table can be found. The SongDAO provides many methods such as get all song of a user, or even obtaining the statistics of songs played per day. Moreover, this class sets basic restrictions which include checking if the user has the permission to alter a song.

- UserDAO

The User DAO is very similar to the Song DAO as it provides all the necessary methods to handle the users and their friendships. If a user needs to register, then a method of this class will allow to add a new user to the database if all preconditions are met. Moreover, this class allows to obtain and modify the user's friendships.

View

- MainView

The MainView class is the manager for all the different views of the server. Thus, as attributes, it has instances of all the possible views the user can interact with: TopFiveView, GraphView, MenuView and ManageSongView. Moreover, in this class important attributes of the views are set. In the constructor of the MainView, the different views are instantiated and other necessary configurations such as calling the *initialize()* method that sets the mainframe where the panels will be placed when going from one screen view to another. This class contains a method for each of the views to be able to remove the previous content and add the new content in the frame. Furthermore, the registration for the buttons of all the views can also be found in this class. In order to update the screen without the need of having to go back and in, the MusicCallback interface is implemented. This interface defines the functionalities for a callback related to music. Whenever there is a need to update the content of a table, it is called.

Controller

- MainController

The purpose of this class is to manage and control the view and the model. This class determines when a screen must be changed or when data must be solicited. To contact the database to retrieve information relating to songs, this class as an instance of *SongDAO*. Also, since the manager monitors which of the views must be displayed, it also has an instance of the *MainView*. The class offer methods to display the views included in the *MainView* and it also assigns the listeners for all of those views.

Database

The database SQL script can be found in the resources folder of the Server package. This database contains three tables. These are the tables with its corresponding attributes and datatypes:

users	
code	VARCHAR(255) NOT NULL
nickname	VARCHAR(255)
email	VARCHAR(255)
password	VARCHAR(255)
PRIMARY KEY (code)	

For each user created an alphanumeric *code* is assigned. All the users have a *nickname*, *email* and *password*. Each user can be recognized with their code as it is unique.

user_user	
code_u1	VARCHAR(255) NOT NULL
code_u2	VARCHAR(255) NOT NULL
PRIMARY KEY(code_u1, code_u2)	
FOREIGN KEY (code_u1) REFERENCES users(code)	
FOREIGN KEY (code_u2) REFERENCES users(code)	

Since a user can be friends with many users, the *user_user* table emerges. This table represents a bidirectional relationship between two users (which are represented with their *codes*).

songs

song_id	INT NOT NULL AUTO_INCREMENT
title	VARCHAR(255)
file	TEXT
min	FLOAT
user_code	VARCHAR(255)
is_public	BOOLEAN
times_played	INT
PRIMARY KEY (song_id)	
FOREIGN KEY (user_code) REFERENCES users(code)	

Each song can be identified with an identification number which is the *song_id*. Each song has a unique id because every time a new song is added, the value is incremented for the next song to be added. The *title* is chosen by the user which contains the title of the song. The attribute *file* contains the songs in text format. The duration of the song is determined by the *min* attribute. Each song belongs to a user, except the default songs, and so their code must be stored in the *user_code* field. The visibility of a song is represented with the *is_user* field. Finally, for each song we store how many times it has been played in the *times_played* for statistical purposes.

stats	
id	INT AUTO_INCREMENT
date	DATE
num_songs	INT
num_minutes	FLOAT
PRIMARY KEY (id)	

For each day, we store the *num_songs* played and the total *num_minutes* of songs played. We also have an *id* for each of the entries, but it is redundant since the *date* is already a unique entry that will never be repeated.

All the necessary privileges are granted to the admin user to be able to access and modified the tables. Moreover, since default songs must be provided at the start of the program, they are manually configured in this script.

Client

Just like the server side, we have also created one package for the server (for communication and information purposes), view, and controller. Each of these packages were created with the intention to separate each of the functionalities.

The most significant classes of the client side are the following:

Server

- **CommunicationServer**

The aim of this class is to control the communication with the server from the client side. With the constructor of this class, a socket is created, and the input and output streams are initialized. This class offers many services for the client that are server related such as getting the user, starting a connection, and getting callbacks. This class extends *Thread* to provide each client a different thread so more than one client can be interacting with the program at the same time. The *run* method allows to select the appropriate action to perform according to the protocol received. The following are a few of the actions that it can perform receive a user, receive songs, delete current user and get friends songs. This class works similarly to the *ClientDedicatedServer* class (in the Server package) as it also uses *DataInputStream*, *ObjectInputStream*, *DataOutputStream* and *ObjectOutputStream* instances to be able to do the necessary operations.

View

- **MainView**

The *MainView* class is responsible to change the views as it contains the main *JFrame* of the client side. This class contains an instance of all the possible views the user can interact with. Some of the methods this class offers are related to displaying different views such as the login view or the register view. By calling any of these types of methods, this class will remove whatever is currently displayed in the frame and display the desired view which can only be possible after repainting, revalidating, and allowing the frame to be visible. Other than being able to move to different views, *MainView* also obtains the necessary tables or shared view components that can be found in any class inside the View package. Additionally, all the listeners from all the view classes are registered in this class. Other methods this

class contains are related to obtaining inputs from the user, such as the user's login, and changing the state of the buttons, for example from mute to unmute.

The piano view functionalities are distributed in three classes:

- PianoKey

The *PianoView* class is used to create the view of the piano keys. The most important method of this class is *pianoKeys()* which returns a JPanel. This method sets the dimensions, white keys, black keys, the octaves that span this set of keys. In other words, this method constructs the graphical interface of the piano.

- PianoSongsView

This class adds more functionalities to the previous class described allowing the user to play a song. Key functionality of this view is the mute/unmute button. The user has the option to play the piano with the song in the background or without it.

- PlayPianoView

The *PlayPianoView* is very similar to the *PianoSongsView*. The main difference is that this view allows the user to play freely and adjust settings. For example, if the user wants to play the note 'Re' with the 'T' key from the keyboard, they can do so by clicking the settings buttons and changing the default keyboard of 'Re' for 'T'. The most special functionality in this view is to record a song. If the user wants to record a song, they must press the record button and once they have finished it, they must press that same button (that will have a pause icon). Once it has been clicked, a dialog will appear (*SaveSongDialog*) to allow the user to set a title, choose if it's public or not, and listen to it again before accepting to save the song or not.

Controller

- MainController

The *MainController* class manages all the other sub-controllers that each of the views has. The controller determines which of the view must be displayed. Not only that, but it offers other methods to obtain tables and panes. Moreover, the inputs that the user enters are managed by this class since they are received by the view. Other methods this class includes are to show dialogs, change icons of buttons, receive information, and assign listeners.

- PianoKeysController

The *PianoKeysController* class is one of the most important controller classes of the whole project as it oversees the logical aspect of allowing the user to play the piano keys with the buttons as well as with the keyboard.

In this class, we have created an inner class called *Note*. This class contains several attributes: *timestamp* (at what time does this notes started to play), *trigger* (indicates if the notes were played with the piano buttons or with the keyboard), *note* (which note it is), *duration* (duration which the note was clicked) and *time* (a static attribute that stores the time of the song) .it stores information of a button click or key pressed from the keyboard to use it when exporting the song.

The constructor of this class initializes a synthesizer. A MIDI synthesizer is used to generate sound. The synthesizer will start playing once it gets a *noteOne* message. The *noteOff* message tells the MIDI synthesizer when the note must stop playing. To play sounds from a piano, we have used a method from the Synthesizer that allowed us to choose what instrument we wanted to play, in this case, a piano.

Moreover, this class contains a HashMap to store which keyboard key corresponds to each note. When an instance of this class is created, its default values are assigned:

Note (Key)	C	D	E	F	G	A	B	C#	D#	F#	G#	A#
Keyboard Key (Value)	A	S	D	F	G	H	J	W	E	R	T	Y

When modifying the notes in the settings the values in this HashMap will be modified.

The *playSong* method plays a song given a String. First, the JSON string is parsed and stored in a *Note* array. For each of the notes, a thread is set considering when it should start playing and its duration. In order to do that

we use threads since more than one note can be played at the same time. An instance of the *Timer* class is used to schedule when each of the notes should stop playing (*noteOff*). There is a similar method that is called the same but not only has a string as a parameter but it also has an instance of a *MusicCallbacks*. The main difference between these two methods is basically how we use them, we use the method without the callback in case we want to replay the song right after it was recorded, which means the string of the song is still stored inside the same class. However, we use the method with the callback in case another class wants to play a song, the callback will tell the other class when the song is done playing, so that's why we created two of them.

This class implements *ActionListener* since we want to know when a piano key button has been pressed. The methods that must be overridden with this interface allow us to know when a key should start playing and when it must be stopped.

Depending on which action the user has chosen, such as record or settings, one or another of the methods will be played. For example, once the user has finished recording a song and they wish to listen to it again (which is an option provided by the *SaveSongDialog*), it will call the *playSong* method with the song string.

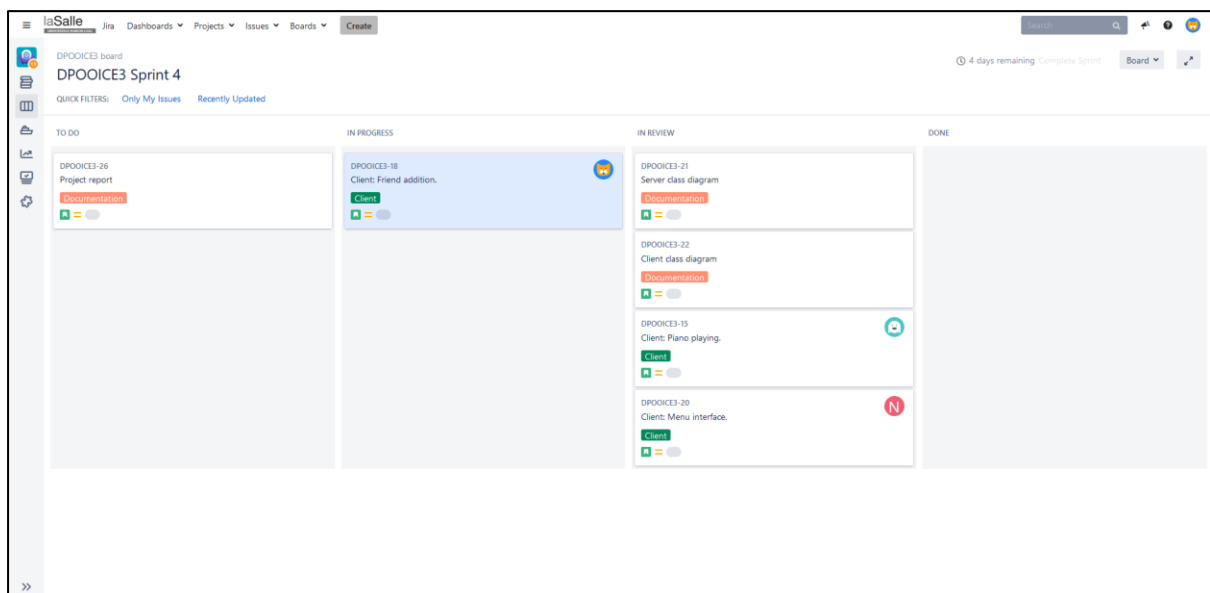
Development Methodology

The development methodology that was followed during the realization of this project was the SCRUM methodology, a subset of the agile development methodologies.

Agile development refers to software development methodologies centered around the idea of iterative development. A team of developers can deliver incrementally, instead of all at once, with greater quality and predictability, and a greater ability to respond to change. This is where the value of the agile development methodology lies.

Scrum expands this and lays the groundwork for this development framework with things like sprints, sprint planning and reviews and a backlog of the different sprints. Sprints are a set amount of time (2 weeks in our case) where the team needs to complete a set number of tasks and deliver certain functionalities. The backlog helps keep track of what functionalities have been delivered and what has been accomplished in the different sprints.

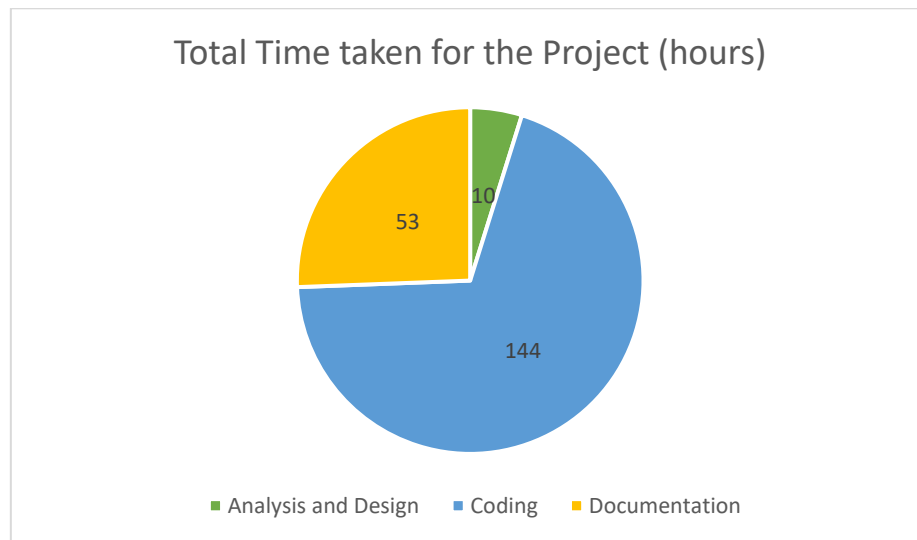
For the development of this project, it was important that the team establish what the goals for each sprint were, and then split up the tasks that needed to be carried out before the end of the sprint. The items/tasks were pre-defined by our professors, with relevant descriptions of what needed to be achieved for a task to be considered done. Once we decided on all the items that were appropriate for the sprint that was about to begin these were placed in the “TO DO” section of the software we used, Jira.



Once we started working on a task, this one was moved to the “IN PRGOGRESS” section. If completed, before moving items to “DONE”, these needed to be reviewed by our piers, so the item is therefore moved to “IN REVIEW”. If the team deemed the item to not be missing any functionalities, or be improperly implemented, then the final step is to move the item to “DONE”. The items in this last section are 100% done, so in order to be moved there, it is crucial they be properly checked by the team.

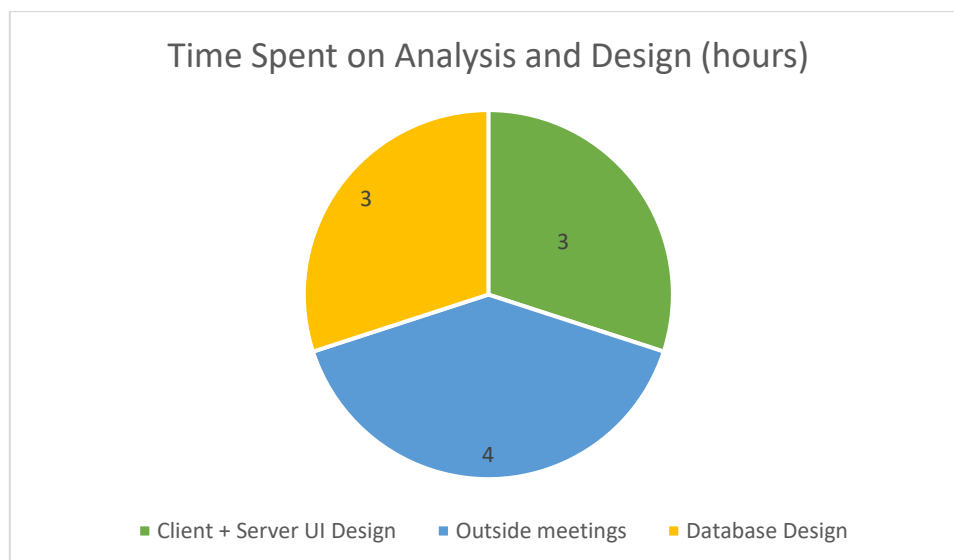
This process happened in parallel. If someone was reviewing a task, they most likely also had a task assigned to them that they would be working on.

Time Costs



Overall, we spent 207 hours on the project. This was spread across a period of 2 months. 33 of these total hours were in class hours, so we dedicated our time to working on it. As can be seen, coding took the most amount of those hours, with the documentation also taking up a significant portion of time.

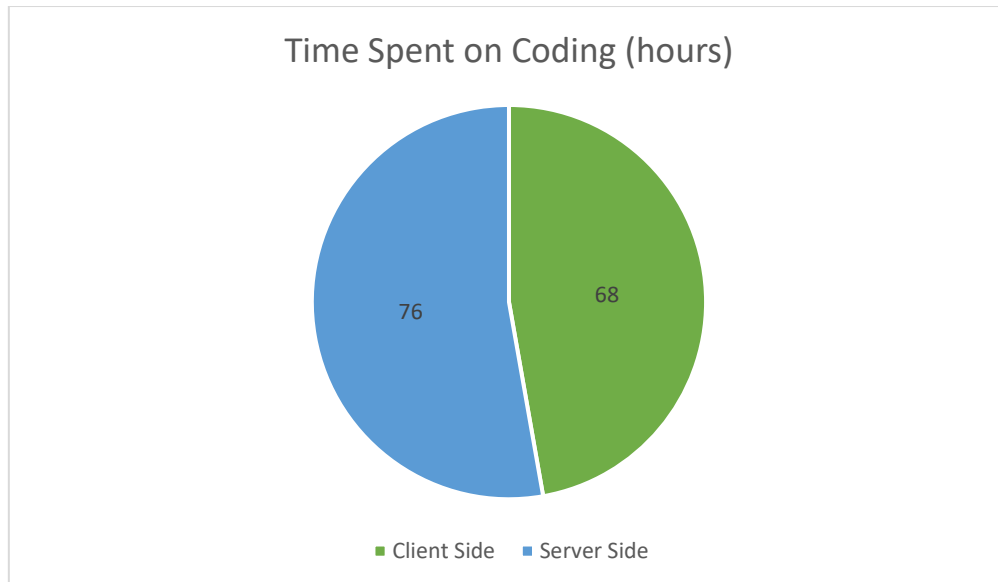
Analysis and Design



For our project, we did not spend as much time on analysis and design as the other aspects of the project. Since we felt prepared to take on this project, we felt that it was relatively straightforward. The client and server UI design only took a short amount of time, as we were familiar with what we could do with JavaSwing. Additionally, the database design was relatively

straightforward, leaving only some datatypes as unknowns. Outside meetings were used to finalize design aspects. Much of this time was class time.

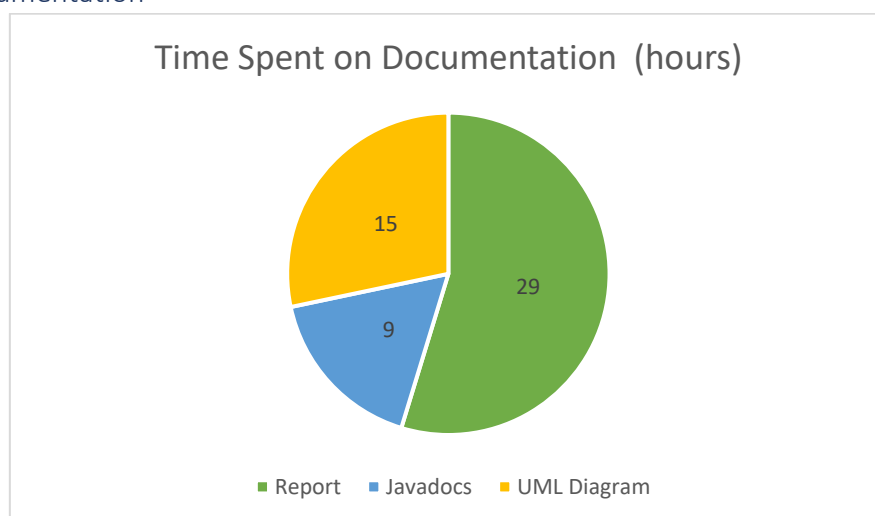
Coding



The thing that took the most time on the server side was not any specific task, but re-working the code so that it could integrate more seamlessly with the other working parts. For example, the UI design in Swing was pretty straight forward, however, we initially made each view a JFrame. So what took more time in that respect was reworking everything into a JPanel, and then designing the navigation between two windows.

The thing that took the most time on the client side was.

Documentation



Although the documentation was a relatively minor part of the report, it still took a considerable amount of time. We decided to go very in depth with our report, and it ended up taking a longer time than anyone expect. The Javadoc took a relatively short amount of time, because we made some comments as we went. However, we still had to go back and update some aspects of it, so we ended up spending a bit more time on it than expected. The UML diagram also took a long time, however, after the initial design time of 8 hours, we had to change nearly everything. This ended up taking up a lot of our time

Conclusion

If we were to redo this project, we would be more mindful of what design patterns we use and how we implement them. For example, in the MainControllers, we made several protected methods to go to the different views (since we use inheritance and the controllers inherit from the MainController in the respective package, server or client), and we called the methods when we needed to go to the different views. We could have made one protected method that returned the MainView, and called the methods inside the MainView class with the returned object.

Another thing we would be mindful of is the libraries that we use. For example, we started the piano design using a library called “JFugue”. This allowed us to very easily create a piano GUI, but it did not match the requirements of the project. We could have continued using it, but we decided against this since we noticed it would have been hard to implement the desired functionalities with this base. We tore down what we had implemented and started from scratch. Luckily, we had not gone too far in the implementation, but it is something to be mindful of when deciding if a library is fit for the job. It may not offer all the functionalities needed, it might have bugs, it might lose support, etc.

Also, we would have been more organized when developing the GUI. For example, initially, when developing the UI, we had each different window as a unique JFrame, and we did not plan on how to connect them. However, once we realized that we needed to communicate with the controller, and move between different windows, we had to reconfigure the entire view package. This alone was not a problem, however, we realized that maybe if we had taken these steps from the start, we could have avoided this problem.

This project was useful to learn how to effectively work in a large group using different time management systems. We all found that using the Jira system was effective and efficient in order to see everyone’s progress, manage the different steps that everyone had to do, and figure out what steps we should take in the next steps of our project. We also learned how to more effectively use github. Although we had used it in the past, in this project, we researched more into using different branches, and saw how that helped keep major decisions. Separated from the main code until we were sure that we wanted to implement them.

Overall, we found that although we made some mistakes in doing the project, we learnt a lot, and would not have known so much if we had not undertaken it.

Bibliography

[DUPLICATE], CLOSING, BANSAL, MOHIT and INFOTECH, SHRIJI, 2020, Closing JFrame with button click. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/2352727/closing-jframe-with-button-click>

BOXES, POPUP, CRONAN, BRENDAN and AZAM, SAQUIB, 2020, Popup Message boxes. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/7080205/popup-message-boxes>

DefaultTableModel (Java Platform SE 8), 2020. *Docs.oracle.com* [online],

DEFAULTTABLEMODEL, JAVA:REMOVING, BERGANDER, MATTIAS, DEGLOORKAR, SHASHANK, NIZET, JB and MAHMOUDI, ABDELHAK, 2020, Java:Removing all the rows of DefaultTableModel. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/10413977/javaremoving-all-the-rows-of-defaulttablemodel>

DEFAULTTABLEMODEL, JAVA:REMOVING, BERGANDER, MATTIAS, DEGLOORKAR, SHASHANK, NIZET, JB and MAHMOUDI, ABDELHAK, 2020, Java:Removing all the rows of DefaultTableModel. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/10413977/javaremoving-all-the-rows-of-defaulttablemodel>

Graphics (Java Platform SE 7), 2020. *Docs.oracle.com* [online],

JAVA, DRAWING, EELS, HOVERCRAFT and CASTRO, RODRIGO, 2020, Drawing a simple line graph in Java. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/8693342/drawing-a-simple-line-graph-in-java>

JFRAME, PASSING and D., DAN, 2020, Passing ActionListener of a List of components in a JFrame. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/14377262/passing-actionlistener-of-a-list-of-components-in-a-jframe>

JTABLE?, HOW and SOKOLYUK, DMITRY, 2020, How do you remove selected rows from a JTable?. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/655325/how-do-you-remove-selected-rows-from-a-jtable>

SIZE, SETTING and EELS, HOVERCRAFT, 2020, Setting JButton size. *Stack Overflow* [online]. 2020. [Accessed 20 May 2020]. Available from: <https://stackoverflow.com/questions/25945439/setting-jbutton-size>

Stack Overflow. 2020. *Staruml Defining Interface Methods And Attributes In The Interface Icon?*. [online] Available at: <<https://stackoverflow.com/questions/6789112/staruml-defining-interface-methods-and-attributes-in-the-interface-icon>> [Accessed 8 May 2020].

Chilakapati, S. and Herron, N., 2020. *Jframe.Dispose() Vs System.Exit()*. [online] Stack Overflow. Available at: <<https://stackoverflow.com/questions/13360430/jframe-dispose-vs-system-exit>> [Accessed 18 April 2020].

Ramesh, G., 2020. *Performing Multiple Actions On One Button In Java Swing*. [online] Stack Overflow. Available at: <<https://stackoverflow.com/questions/12000641/performing-multiple-actions-on-one-button-in-java-swing>> [Accessed 14 May 2020].

Stack Overflow. 2020. *Actionlistener From Controller Doesn't Trigger*. [online] Available at: <<https://stackoverflow.com/questions/48303440/actionlistener-from-controller-doesnt-trigger>> [Accessed 13 May 2020].

Stack Overflow. 2020. *Staruml Defining Interface Methods And Attributes In The Interface Icon?*. [online] Available at: <<https://stackoverflow.com/questions/6789112/staruml-defining-interface-methods-and-attributes-in-the-interface-icon>> [Accessed 8 May 2020].

Svensson, P. and Skeet, J., 2020. *Javadoc Interface Comments*. [online] Stack Overflow. Available at: <<https://stackoverflow.com/questions/9234571/javadoc-interface-comments>> [Accessed 17 May 2020].