GHMF Developer Quick Guide

For W14 Development Team

**Introduction:**

The following document was constructed for the sole purpose of informing the next development team of how the java project works as a whole. There are “2” main parts to this project - one of which is the choreographer, the other is the simulator. The design of the project is such that these two main components are closely related and appear at the same time when the program is started (hence “2” despite it being one program).

The **simulation** part of this program is designed to visually reproduce the contents of a control file, as close as possible, to what the real musical fountain in Grand Haven would display. In addition to the documents that will be provided in the project, Youtube videos are available for further viewing to get a better idea of how the fountain operates.

The **choreographer** part of this program is designed to allow a user to modify a control file using GUI components. The changes to the control file should be reflected in the simulation, and then a user can save the changes to the control file. The combination of the simulator and the choreographer allows the user to create, edit, and view the control files that can be given to the Grand Haven to play on the real fountain.

**Overall Design of the Java Project:**

The project itself was developed using the Eclipse IDE. Packages were created to house the different aspects of the project with the attempt to use the Model View Controller Design pattern.

**Before continuing onto how the current details are handled in the Java project, we recommend reading documents that help explain the fountain as a whole** (GHMF Fountain Command Language Reference, Project Scope of Work, Layout Diagram of Fountain, etc.). Note that some of these are documents from the previous group that may or may not be fully correct or up to date. When it doubt, asking APEX (Terry) will likely get you the correct information.

The “backend” of this project revolves around the use of a control file, a music file, a map for colors, and beatmark file. These files are fairly modular (a control file can be swapped out if necessary, as can the color map, etc.). The control file contains commands to occur at tenths of a second, the beatmark file contains marks at tenths of a second, the music file contains the music to be played, and the color map contains the 32 colors available to the user. For more detailed information about the control file and its design (as it is essentially the main piece of this program that is dealt with a lot), see the latest FCL revision. At the time of writing this, it is at FCL-03.00.18.

In regards to the control file, the core piece of the model is a map where the key is an integer that represents the time index (in tenths of seconds) and the value is an ArrayList of fountain control words (FCW). It functions similar to any other map with a <K,V> structure, but V is actually an ArrayList of all the different command signals to occur at time K. This map of <Integer, ArrayList<FCW>> is the control file in modifiable form. Note, the fountain runs on signals. FCWs are signals to turn on or off the specific water or light modules, and the map of <Integer, ArrayList<FCW>> contains a list of such signals at specific times. This structure can be seen in the class Timeline.java, which is a core part in managing the model. In addition, this data structure is what ultimately tells the simulator what signals occur at what times (so what should show up), and it is what is modified when the user makes changes in the choreographer. As stated before, this is essentially the control file in a format for use for the programmer.

Numerous classes are defined with simple accessor/mutator methods (or perhaps very simple logic) that are used in the view and model. For example, FCW is a class with instance variables for the Address and the Data, while Fountain is a class that is used to group the different water modules. These are fairly straightforward.

There are multiple classes with singleton design patterns that are used throughout the program. For example, the LagTimeLibrary is used to look up the appropriate “lag times” for commands - usually water commands - to know how far back they should be saved in the control file. This is because it takes time for the physical water mechanics to occur in real life, so to have the command actually look nice it needs to be moved by a small lag time when saved in the control file. Another class with a similar pattern is FCWLib. This class is used to “look up” commands, both from a numeric value (address-data) and a named label (“bazooka”). Given one, it should be able to give you another. This is particularly helpful for when an action in the choreographer gives a name for what control was edited and can then insert the correct FCW (AAA-DDD) into the model - the map that was previously mentioned.

Lastly, there is an I/O package that is used for saving and loading of files. This is relatively simple as there isn’t anything too ground breaking in terms of I/O going on. Saving the map of <Integer, Arraylist<FCW>> is the important piece of this package as it needs to correctly save the commands at the specified time (with a lag time if appropriate).

The FountinSimController is the class that in essence runs the whole simulation. It is fed the FCW’s at the current time and then displays them on the screen. The drawFCW method accepts an arraylist of FCW’s for the current time and then sends the commands to the correct addresses. It is very easy to understand overall, but in it’s present state it is very bloated. There is not a clear distinction of model/view. There is one method per water feature that animates the levels. The sweepers are the most confusing of all of the water features since further animation is need for the rotation animations.

The sweepers have four methods for each set of left/right sweepers: draw, sweep, oscillate and smooth. The draw method is like all the other ones and is for controlling the height. The sweep method causes the sweepers to sweep from left to right. Oscillate causes the sweepers to act like a fan. Both of these methods can be used to hold the sweepers at the current position of both inputs are the same. Smooth sweepers is meant to transition the sweepers from their current position into the next sweep motion. It is completely written except for one exception. It currently cannot figure out the current position of the sweeper. JavaFX provides no way to have the angle found out. The only possible workaround is to create a virtual line that changes a number based on speed and direction of what the simulated sweepers are doing. If the angle is able to be found, the only thing that needs to be changed in the smooth Sweeps is double Angle is the current position.

**Where to Start**

Based on our experience with the project thus far, there seem to be a lot of issues that are hard to find and fixed due to the nature of the design. Despite the attempt at the MVC design pattern, there are quite a few areas where this pattern was not followed. As a result of the deep integration, it is hard to assess whether or not the use case is failing at the back end / model level or at the user interface level.

We recommend working from the ground up. Ensuring the validity of the back end functions being used will isolate problems towards the UI end of the project, making finding and fixing errors easier. Included in the files given is a list of all of the .java files (classes) that are in the project. Classes with information in the ‘Additional Information’ column should be given a closer look at first. Some back end classes specifically to look at are FCWLib.java and LagTimeLibrary.java/LagTimeTable.java. Both use text files in the resource package to perform the look ups. Both of these files should be double checked with Terry and the latest revision of the FCL to ensure that lookups will be reading in the correct values in the first place.

From here, work can then be done on controllers, which are a part of the large functionality of the user interface. Due to their size, refactoring is recommended at your discretion - either after or before attempting fixes to the relevant portions. In general, isolating particular problems or functionality while testing it with code or functionality that is working well is obviously recommended. Furthermore, getting pre-existing functionality high priority features fixed first is a much better use of time than implementing new features that are not as desirable or mainly for convenience. Consult APEX to get an idea of what features/functions are of the highest priority.

There are several of us that are willing to meet with you and help you in whatever way we can to get you started. The hardest part about this project is understanding how everything works. Send us an email and we can set up a time to meet with you as soon as possible.

Matthew Offringa - offringm@gmail.com - Worked on GUI & Simulation

Cameron Ohrt - ohrtc@mail.gvsu.edu - Worked on backend

Joseph Kazakos - joe.kazakos@gmail.com - Worked on backend

Danny Selgo - dgselgo@gmail.com – Light Timeline

Quick Instructions

Selecting boxes on timeline - press the shift button and then left click (hold) and drag the mouse. Then you then can right click your selection to access fading, and copy/paste.

To put color onto the timeline, select color from color palette and drag it onto the boxes. To change a color in the color palette, pick a color using the color picker on left side of a palette. Then you can right click any of the bottom 16 boxes and change the color.

Saving as a GHMF bundles all the files into a zip file. These can then be opened using the program. Also, to easily open files, you are able to drag any valid file onto the program to open it.

Other hotkeys are available (see the designated keys on the menus on the top of the screen).