Alex Crease
Mark-Robin Giolando
Victoria Preston

MusicSwAPPer: Software Design Fall 2013

Initial Project Proposal: October 25, 2013

With the sprawling number of music sharing websites, downloaders, and applications available individuals can personalize their music preferences. Gone are the days of, "I like rock." Now, its, "I like rock – but the alternative, obscure, heavy metal, 80s-esque kind…oh, and bagpipe music." Even amongst friends, musical tastes vary widely. How many times are we asked or told to listen to that newest hit, or to that obscure indie band with this one hit wonder….and then never do?

We are proposing to make a web-based platform for sharing music with friends directly and dynamically – at the cost of listening to what friends have shared with you. Roughly, the system would allow a person to develop a playlist for each of their connections within the app. The playlists can only be shared or developed however, if the person has listened to the playlists provided by their counterparts. Ideally, there would be a way to track how often particular songs are shared, and to track the 'top sharing lists' which could be a jumping platform for new users.

Within the scope of Software Design, the minimum deliverable will be a downloadable module in which a user can share links to other users. This module would provide the user the ability to open links directly on the application.

The application will have at first the basic format of a number of users with an option to send a link of music to each user. If the sender has not watched or listened to a music video from the person they wish to send the link to, their option of sending a link is blocked until they have watched a video from the recipient. The beginning setting allows for links to be sent to kickstart the process. We could also experiment with the number of songs a user can send out before having to listen to a shared song, with the theory that not every user will send recommendations at the same rate.

As the route of link sharing will be used, then text and URL handling dynamically needs to be researched, and there would be less concern for python's music modules. This would allow for a user to send a youtube link to another app user. If that link were clicked, the program would mark that the user had seen or heard the link, and would thus allow for the user to send a link back in

turn. This would allow us to focus in more on the actual development of the app as opposed to the sending, receiving, and playing music files, and could serve as a minimum deliverable should the purely music portion of the code fail.

In terms of creating a successful project, the biggest obstacle will likely be in interfacing existing Python modules with self-designed code, and basing it off a web application. The number of new concepts would be large, and there would need to be a significant amount of self-direction. Programming in python for use on the web is one example of these concepts; the python wiki provides documentation on communicating with the web server using CGI, which sheds some light on the subject, but we will need to figure out how to make the application and build it for use on the web while coding in Python.

However, such an app would be both very useful and fun as it encourages communication and music sharing in a way that just saying “Hey, you should listen to this song” does not, as it allows people to explore new music but also suggest music to their friends, with a price; that they have to listen to songs their friends have suggested.

Preliminary Design of the swAPP: November 8, 2013

The goal of the swAPP is to create a social interface of music sharing amongst networks of online friends and communities. The overarching mission is to make it easy for someone to share online links with others in a fun social environment, while expanding personal online repertoire by receiving and interacting with links from others. To this end, the swAPP must be user friendly, interface with other users, store and transmit data between users.

Initially, the swAPP was considered to be a web application, rather than a python module. After considering the time it would take in developing a javascript shell to make this a reality, it was decided that the swAPP would be developed as a downloadable module written primarily in Python. However, should we decide to take it to the next level and make a web application, the web.py web framework would come in handy, which makes it extremely simple to make a website using code from Python. Http://webpy.org/ provides documentation, tutorials, and examples of the use of web.py that we can use to turn our project into a web application.

There are two main functions of swAPP - ‘sharing’ and ‘viewing.’ Users are asked to build a network of link friends, and send a corresponding number of links to the number of links they in turn use or listen to. Below is a draft of the set up of the GUI. Note that the left side of the page consists of mostly user inputs, including an input for a link, a person/email address/username to share with, a title, and a message that the user can edit. This is the “sharing” side, in which the user can share videos with others, and look at what they have shared in the past. On the right is the “View” side, in which the user can see and watch videos that other users have shared with him or her. On the bottom, the user can see how many shares he or she has left. This value increases when the user watches videos and decreases when the user shares videos. As an added feature, one can share the links they’ve watched via Facebook or Twitter. This would be a maximum deliverable feature however. The “see history” button goes to a separate page of all of

your “shared with you” and “shares” lists.

On the backend, the elements on the application can be sorted into ‘Inputs’ and ‘Output’:

Inputs	Output
Username of person to share with	Data to send to another user
URL/Link	Sharing History Box
Editable Description and Title	Editable Description and Title
Social Media Sharing	Shared History
Share Button	Video/Link interface
History Button	Full Window Past History Display
	Points Display

Inputs:
Username of person to share with - in order to share across other modules, a network will need to be developed between users in order to transmit the data. For personal storage, a dictionary of name: account, email will be kept as a global variable.

URL/Link and Editable Description/Title - using urllib, htmlparser, Beautiful Soup, and some other modules, some screen scraping might be done of the link in order to provide sample text for the

editable descriptions. This hinges primarily on the legality of doing this with certain websites. Ultimately, these same packages might also allow for some data scraping like connected link choices to be logged and suggested via the application.

Share Button - will be used as an indication that a ‘Share’ is ready to be sent to a user, it will update the points and sharing history features.

History Button - will be used to open a separate window which is a full list of links viewed from the shared list.

Social Media Sharing - will allow the user to post to an outside social media source for mass sharing, would not count as a point subtraction.

Outputs:
Shared History Box - a list of tuples of link, who shared with you, date. Will send links in the display to the separate history logged once it is registered that this link is ‘used.’

Sharing History Box - a list of tuples of link, who you shared with, date. Will update with the Points Display.

Points Display - for now, a simple number display of (#of users connected to + History - Shared), which answers: how many can I still send?

GUI:
For the GUI we are using tkinter to create the layout we desired above. Currently, we are partway through Lundh’s An Introduction to Tkinter as well as tutorials on zetcode and effbot. The basic organization will be through grid geometry.

Deliverables:
The minimum deliverable will be a downloadable module in which a user can share a link to another user’s app. Users can open the music links directly off their application to watch and listen to them.

At the highest level, the team would create a web application that, in addition to the minimal deliverable, would allow users to share and receive links. The full intent of the application is to build a network of strangers connected by musical tastes and shares.

Status:
Our first step was to create the logic for the program itself; that is, creating a system that will log “entries” and “exits” - the passing links - into the program; corresponding to the shares that have been made and the links that have been shared. We then wanted it to function alongside the interface and keep track of the “history” of inputs and outputs.

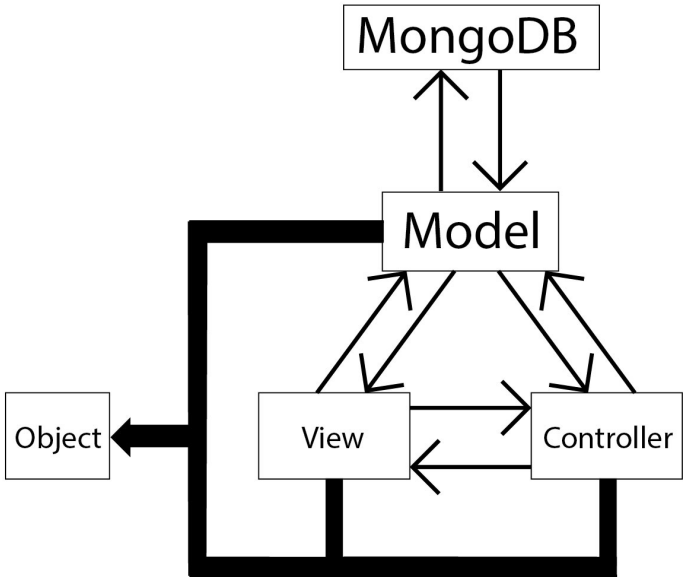
Design Refinement: November 22, 2013

As development continued on the project, the proposed design had to be adjusted in consideration of developing a Model-View-Controller system. Previously, the intent was to create a single database of collections associated with each user. The time to access the appropriate data for every user, and display it accordingly was challenging to conceptualize in code, and to display on the command line or GUI. Currently, the module now operates with several smaller databases of information, which is tagged by user. The information we store includes a share log, being the links that the owner of the module shares with friends, and a viewer log, being the links that the owner of the module receives from others. A collection of user points accumulated from shares and views was also created.

The database system that we will be using in the prototype is MongoDB. Redis will be considered in a final iteration, but MongoDB provides a well documented and simple database access system for python. The largest concern with this choice is in the speed of access using MongoDB is data structures become too large. If this becomes an issue, a redis database system will replace the MongoDB system.

A function is also added to the GUI to initialize the system by accessing the data for the owner of the module, and displaying it appropriately on the screen. As the owner of the module receives data and sends data, the GUI will display the updates upon request in the prototype. The GUI itself will be developed using tkinter, as accessed from GUI() in the swampy package. This was decided above using the GUI Creator system QT, for it's ease and compatibility with python. In a final iteration of the project, QT may be used over tkinter for its beauty and control of GUI design.

The UML diagram, shown to the right, represents how our model communicates. Both the view and the controller are operated by the user via the GUI, in providing function blocks like buttons and entries, and viewing stations in the form of canvases.



The language that we will use for our system revolves largely around the actions of links. A link can be 'shared', and when it enters the system, it is called a 'share.' A user can share a link with another user. A link can also be received, describing a link that another user receives from a

friend. A link that is clicked and viewed in the application is considered a 'view.' Users that are connected are called friends. The system allows a user to see their 'share history,' a list of links that they have sent to friends. It also provides a view list, which provides all new received links from friends.

Over the break, the team intended on increasing MongoDB functionalities in the user-sharing space, and encasing current functions fully within classes. An additional module, Beautiful Soup or others, for displaying webpages would also be investigated. Following break, improvements to data structures and networking would be the primary focus until the demo period. Our project is located at https://github.com/vpreston/MediaShareProject.

Final Report: December 17, 2013

The terminating product has the capability to send links between module users, access and write to a database of link sharing and views, and open/display links within the application itself. The final product is seated nicely between out minimum and maximum deliverables claimed at the beginning of the project, and ultimately taught us a lot about databases, server/client relationships, and gui development.

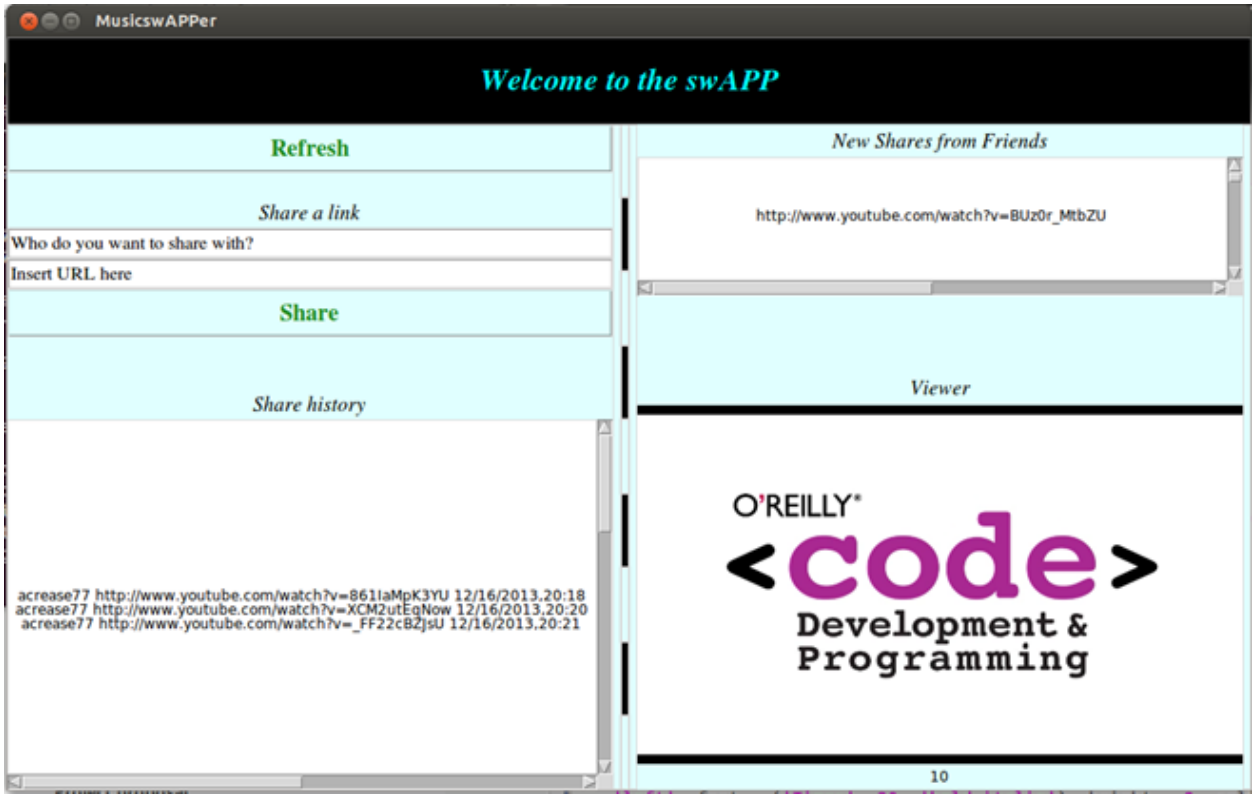
Final Refinement:
Previously, the intent was to create a single database of data associated with each user and access the appropriate keys for each instance of our module. The time to access the appropriate data for every user, and display it accordingly was challenging to conceptualize in code, and to display on the command line or in the GUI. The module now operates with several collections of information, which are tagged by user. The information we store includes a share log, being the links that the owner of the module shares with friends, and a received log, being the links that the owner of the module receives from others.

The database system that we decided upon using in the prototype was MongoDB as MongoDB provides a well documented and simple database access system for python. This choice also led us to use the internet host MongoHQ over redistogo for third party storing and access.

Our system's language largely revolves around the actions of links. Links can be shared from user to user, and a counter keeps track of the difference between the number of links a user has viewed and the number of links a user has shared. The system allows a user to see their "Share History", a list of links that they have sent to friends. It also provides a "Received List", which provides all received links that a user has not seen yet.

Since the last design refinement we increased MongoDB functionalities in the user-sharing space and organizing our code into classes to make it easier to read and to debug. We used a combination of a download youtube videos script and gstreamer so that users could watch youtube video links directly on their GUIs. We have also implemented networking between multiple computers so that the users can successfully receive and send links. Our project is

located at https://github.com/vpreston/MediaShareProject.



Our final GUI display

Our team achieved somewhere in between our minimum and maximum deliverable. We achieved our minimum deliverable fairly early. We were able to create a GUI, interact with MongoDB and use it to send and receive links to other users, and using website parsing we were able to get information from the website that was shared. We then created a counter that would subtract one every time a link was shared and add one every time a link was watched. However, at the time that we reached our minimum deliverable, we had not really tested the networking portion yet. A lot of our minimum deliverable is incorporated into our final iteration, except we added the ability to click on the video links and have them download and play on our GUI. We also added a “Sign Up” and “Sign In” interface that opens up before our main GUI opens up to allow for users to make accounts, log in, and share links with other users.

Our team decided to stick with a downloadable module for simplicity and so we didn’t have to deal with interfacing with the web, and also decided to stick to TKinter because of its simplicity. We have two displays, one that displays the links that you have shared with other users, and another that displays links that other users have shared with you that you haven’t watched yet.

Although we did not reach our maximum deliverable, we are satisfied with the outcome of the project. A large portion of our time was spent understanding the various things incorporated into this project, including GUIs, databases, and networking. We feel that we would be able to redo the project in a much shorter time span with the knowledge that we have, and improve a variety

of features about our code to make it more user friendly.

The design as a whole was divided into discrete chunks, and from a development standpoint, the definition of the GUI and functionalities as modular was the best design decision. With the ability to work on little functions at a time, it was an easy matter to set small, attainable deliverables that led to the final product.

Dividing the design into discrete chunks also worked well from a robustness standpoint. If there is an error or a bug somewhere in the code, it is easy for us to find as the segment that the bug is in will be the one not working. This also allowed us to run tests on multiple segments at once as a bug in one segment would not prevent another segment from running except for a few exceptions.

If there was a 'next time,' we would move the development in databases closer to the front of the project along with the GUI. The databases provide to be more complicated and detailed than we had originally planned, creating difficulties in later developments of the project. Additionally, we would have spent more time prototyping in classes rather than develop the initial project in classless code and have to go back to convert the code into classes.

We divided the work based on modular behaviors. For example, while one person would work on networking, another would be working on video viewing. This allowed us to tackle challenges in the most time efficient way possible. Once we had a behavior working we could incorporate it into the project as a whole. One way we were able to do this was getting the GUI done early and by designing the GUI to easily accept the behaviors as they were completed. The times we weren't able to divide up the work among individuals as easily came when we had to test the networking as this required more than one computer. Much of the testing for this came from tests run during group meetings.

Some of the most difficult bugs came with the use of databases. Understanding the new language of data storing and access had a considerable learning curve, and because databases and collections are hysteretic, development could be difficult if previous tests were not fully cleared. Accessing and writing data to the databases keeping in mind that many distinct users may be using the database at a time was also a considerable design challenge. Many hours were spent tracking through code to make sure all collections had a similar enough format to reference themselves, and that functions were updated to the latest format of each database, which was fluidly changing to accommodate the design requirements of a multi-user application. In order to debug, many test statements were built into the code to find where the code was initiating (or not), what the data string being accessed looked like, and marking the code execution given an initial case. The edge cases were also tested to make simple debugging easier - new user versus old user, no shares, many shares, no views, many views.

What could have made the bug extermination faster would have been a simple consideration for the format of the databases. Once the development script was written in such a way to clear the databases before every new test, debugging became significantly simpler once there wasn't huge data strings to read through and process. More group debugging as a whole on the database side would have also made things generally faster for these particular bugs. Using a different development system (outside of multi collections for data types) may have also been a more efficient way to go for debugging. Rather than separate by data type, perhaps separating

by user would have been a simpler way of going about it.

Conclusions

Overall, this project dealt with a variety of different subjects, including networking, GUIs, databases, and web scraping. Our project was an interesting exercise and introduction to the world of module and application development, with new skills applicable to developing interfaces and web applications down the line. It was also an interesting experiment in data structures, data storage, and data processing, which has sparked a variety of ideas to efficient access, and development organization. On the whole, the final project experience was successful in product and learning goals.