- Software Design Document -

Version 2

January 26, 2024

DENDRO-DAWGZ

|Growing A Brighter Future|

- Team Members -

Nile Roth

Niklas Kariniemi

Zachariah Derrick

Asa Henry

- Sponsored by -

Prof. Andrew Richardson, SICCS/ECOSS

Prof. Mariah Carbone, ECOSS

Prof. George Koch, ECOSS

Austin Simonpietri, ECOSS

- Team Mentor -

Tayyaba Shaheen

Table of Contents

1 Introduction

Dendrology is the study of trees, and it continues to be one of the most overlooked sciences. With the current rates of deforestation causing concern around the world, dendrology studies are more essential now than ever. Millions of acres of forest land are torn down every year for the establishment of businesses, farms, housing, and sometimes just for their natural resources. Tree physiology and ecology play a big role in providing the population with the understanding of tree's significant involvement in the balance of life. Our project focuses on easing and supporting research projects like these. More specifically, we are alleviating the processes of installing and obtaining data from dendrometers.

A dendrometer is a data-logging instrument that measures the changes in diameter growth over time. They are generally used on the trunks of trees, but can be used on many different types of plants and expanding objects. The data retrieval process from these devices in trees is very laborious due to the fact that the data is transferred through a short cable that connects to USB. This implies that a laptop must be transported up the tree in order to successfully obtain the data. This perilous process is not only physically demanding, but also puts costly equipment in jeopardy. Portability is the essential feature that this system lacks. To assess the issue we plan to implement a mobile application with an easy to navigate user interface. Users of this application can use their android phones or tablets to easily obtain and observe the dendrometer data. This application also allows for the analysis of multiple dendrometers' output simultaneously.

The simplification of this process will not only ease the lives of those who retrieve the dendrometer data, but will also help improve and increase the flow of data needed for research projects. Many research projects at Northern Arizona University (NAU) and around the globe depend on frequent data retrievals from these dendrometers, meaning the process should be simple and efficient. When we make this task less of a hassle, there will be an increase in visits to collect dendrometer data, overall leading to a larger existence of research relevant statistics.

We are currently in the midst of the implementation stage of this project. We currently have an application with the basic functionality of obtaining and displaying the dendrometer data. Our current research involves the process of merging data files, and the ability to handle and display these merged files. In the near future we will begin working to include statistical analysis to our application. In this document we will go over our project's overall architecture and the design of each major module.

2 Implementation Overview

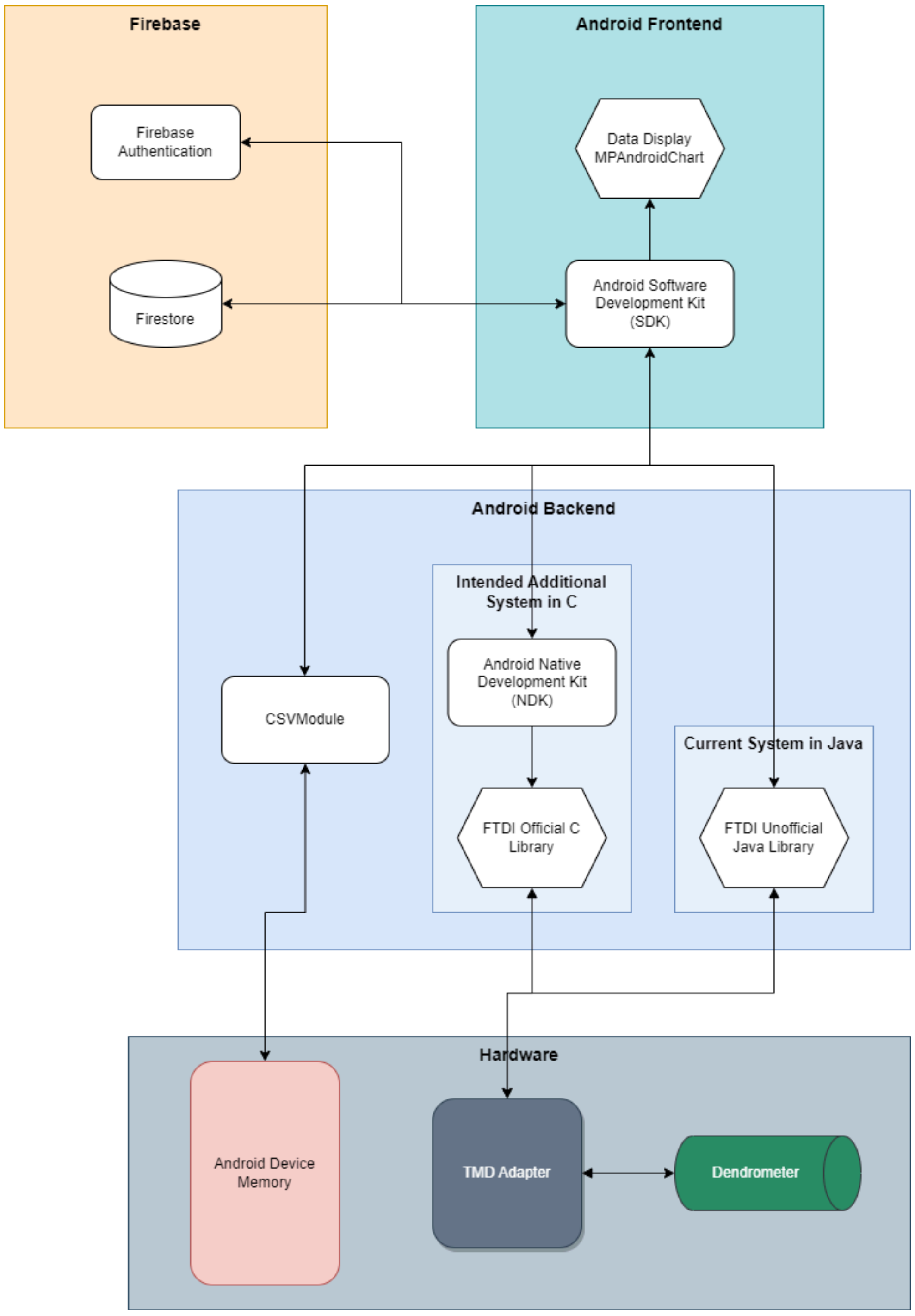
Our intended solution to solve our clients' problems is an Android mobile application that utilizes FTDI chip libraries to communicate with pieces of hardware - a TMD adapter and a TOMST point dendrometer - and display the graphically retrieved data to the user of the application. Our application will additionally utilize Google Firebase in order to export data to the cloud and share it with other users.

The overall approach we are taking with this application resembles multiple software design patterns, namely the Hardware Device design pattern and the Producer-Consumer design pattern. Our application will be communicating heavily with an external piece of hardware, and afterwards it will be consuming that data and converting it to a user-readable format.

The frameworks and packages that will be key for our application include the Android SDK and NDK, the FTDI chip library, Google Firebase, and MPAndroidChart. The Android framework will be a critical component of our application, because the application itself will be built upon this framework, utilizing the SDK for user interface and backend components, and the NDK for native device communication - such as communicating with the dendrometer. We will be using both the official C FTDI chip library and the unofficial Java FTDI chip library in order to directly communicate with the hardware. Google Firebase will be used to export and store user data, authenticate users, and share data with other designated users. Finally, MPAndroidChart will be used to graphically display the data to users of the application in conjunction with the Android SDK. In total, our mobile application will be written predominantly in Java and Kotlin, with a moderate amount of C for communicating with the hardware.

3 Architectural Overview

3.1 | Architecture Diagram



3.2 | Architecture Discussion

The architectural pattern we are modeling our Android application on is a layered model-view-controller architecture (MVC). In our system we have several layers that help separate distinct domains within the software and contain specific responsibilities, including the UI/Frontend layer, The Domain Layer/Backend, the Hardware layer, and the Cloud layer.

The frontend is responsible for providing the entire user interface for the mobile application, which is done through the Android SDK and MPAndroidChart. The key responsibilities of this layer include providing the interface that begins the dendrometer data download, the graph display of dendrometer data, selection of dendrometer data to be displayed, and other typical interface functions like settings and app-specific options. This layer communicates with the backend frequently, through the Android SDK and NDK, typically to initiate dendrometer reading, or CSV file manipulation. This layer also communicates with Firebase at the cloud layer through the Firebase API to log users in, to store and retrieve user specific dendrometer data, and to share data amongst users.

The backend is responsible for most of the heavy lifting within the application, communicating with the hardware layer frequently through the FTDI chip library both in C and Java. We plan on moving as much Java code over to C as possible. The backend sends commands through the FTDI API to the TMD adapter, which retrieves data from the dendrometer and sends it back to the application. The backend translates this data into human/graph readable CSV files and stores them locally on the device. Manipulating the CSV files and loading them into the application using the CSVModule, and reading data in from hardware are the key responsibilities of the backend.

The cloud layer consists entirely of Google Firebase services, including Firebase authentication and Firestore. The key responsibilities of this layer are to communicate with the frontend in order to log users in, store user specific data, and share data between users. Firebase handles the specifics of authentication and the database, and our application communicates with it through the Firebase API.

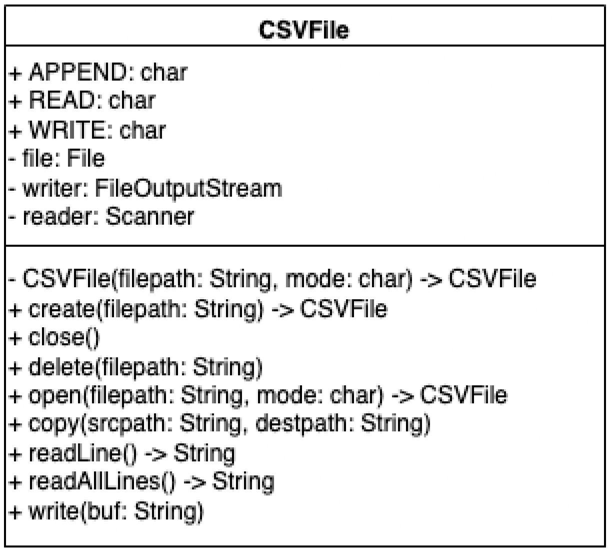
4 Module Interface Descriptions

4.1 | CSV Module

4.1.1 | Description

Because our application will allow the visualization of one or multiple datasets from TOMST dendrometers, a module to handle reading and writing CSV files is required. The module will be responsible for reading a CSV file into memory to be displayed, writing mutated data to a new file or overwriting a file, as well as copying a file. In the event the data is mutated or copied, the module will create a new CSV file and write the data.

4.1.2 | UML Diagram



```
CSVFile
+ APPEND: char
+ READ: char
+ WRITE: char
- file: File
- writer: FileOutputStream
- reader: Scanner

- CSVFile(filepath: String, mode: char) -> CSVFile
+ create(filepath: String) -> CSVFile
+ close()
+ delete(filepath: String)
+ open(filepath: String, mode: char) -> CSVFile
+ copy(srcpath: String, destpath: String)
+ readLine() -> String
+ readAllLines() -> String
+ write(buf: String)
```

4.1.3 | Public Interface

Making up most of the CSV Module is the “CSVFile” class which provides other modules a concise way to manipulate datasets without having to interact with Java’s file IO interface directly.

The CSVFile provides two ways of opening a file for data. The first way to open a file is via the public, static “create” method which wraps a call to the *createNewFile* method on Java’s File class. The *create* method accepts a string specifying the path at which to create the file, as well as the name of the file to create. Upon successful creation of a file, the *create* method will return a CSVFile object through which a module can interact with the new file. By default, the *create* method opens a file in “write” mode, since no content will exist in the new file.

The second way to open a file is via the public, static "open" method which handles opening a reader and writer to interact with a file on the system. The *open* method accepts a string specifying the file path, as well as a mode to indicate whether the module intends to write or add to, or reader content from the opened file. Upon successful opening, the *open* method will return a CSVFile object through which the module can interact with the opened file.

Once a file is created and or opened, the public "write", "readLine", and "readAllLines" methods become available to a module. The *write* method allows the writing or appending of content to an opened file. The *write* method accepts a buffer string, of any length, and writes the contents to the opened file. Whether the content is written or appended, depends on the mode specified when opened; otherwise, the content is written if the file was created.

The *readLine* method accepts no parameters, and returns the content from the last newline character (\n) read to the next newline character. Similarly, the *readAllLines* method reads all the content in a file, and returns contents as a string. The file must be opened in "read" mode to use these methods.

For the event a module needs to make a copy of a file, the CSVFile provides a public, static "copy" method. The *copy* method takes the path to the file to be copied, and the path to which the file will be copied. The *copy* method takes care of opening the source file, creating the destination file, and moving content from the source to the destination.

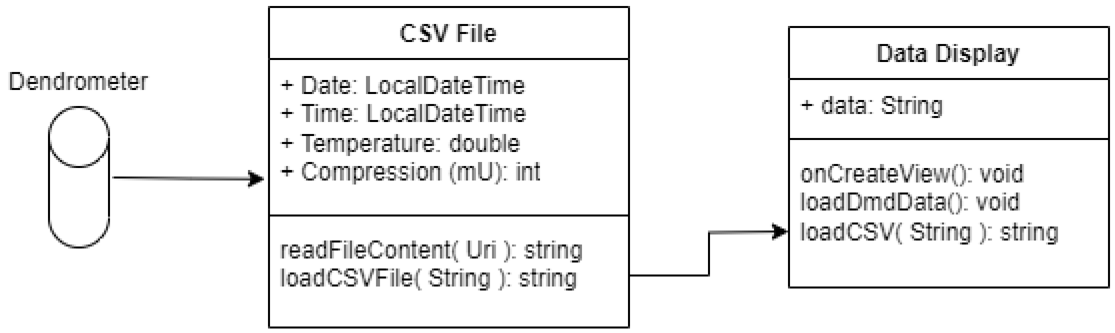
When a module is done working with a file, a public "close" method is provided. The *close* method takes no parameters and returns nothing; this method is solely used to close the reader and writer when a module is done working with a file.

4.2 | Display Data Module

4.2.1 | Description

Creating a visual representation of the data collected requires there to be what can be understood as a canvas onto which data signifiers (i.e. points, lines, etc.) can be drawn. The data collected from the dendrometer comes in the form of large csv files, and must be translated into a useful line graph. This module is the most essential part of our application because it makes up the biggest and most used UI component of our software. Users must be provided with consistent and precise visualization of their data sets. In order to accomplish this transition of raw data into useful statistics, we utilized the charting library MPAndroidChart.

4.2.2 | UML Diagram



4.2.3 | Public Interface

The majority of work done for this module is handled by MPAndroidChart. This library is widely used for the organized representation of data, and has the ability to implement interactive, appealing charts and graphs. As stated in the previous module, the data collected from the dendrometers are brought into our application as CSV files. More specifically, each line in the data file corresponds to a single measurement taken by the dendrometer. A single measurement consists of a reading of the date, time, temperature, and compression (tree growth). After this data is obtained from the file, methods in the GraphFragment.java file utilize many of MPAndroidChart’s functions to transform the data into a precise graph.

A method named “onCreateView" is executed on the click of “view data”. This creates an empty X-Y plane for the line data to be displayed on. The “LoadCSVFile” method uses a for loop to scan through the file entries and create an input set that is recognized by the charting library. This method utilizes the sub-functions processLine and readLine. readLine simply reads a line of the CSV and returns it in a string format. We can input this string as the parameter of processLine to parse the string and extract its data.

At this point we can simply use MPAndroidChart’s setData function which outputs the set of data onto the already created empty graph. This is done by the private void “LoadDmdData" method which also incorporates an animation and initial setting of view for the user.

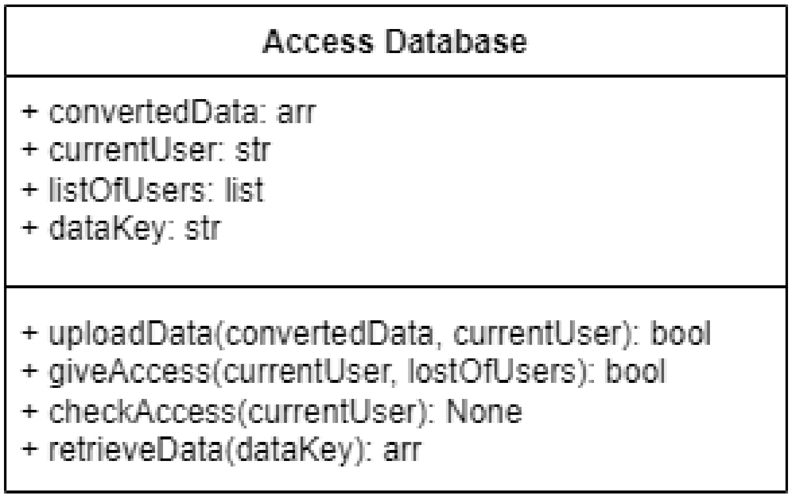
In some instances, users may desire to view multiple data sets on a single graph. This allows for an efficient comparison between two or more dendrometer readings. We are currently finishing up final touches on a new method “mergeCSVFiles”. This will be called upon to combine the components of multiple CSV files. LoadCSVFile handles the input of a merged file, and is able to process the lines of more than one file into multiple data structs. The reading from MPAndroidChart is the same process, just called once for each dataset.

4.3 | Database Module

4.3.1 | Description

Our mobile application will need to have the ability to upload data to a cloud database. This is the reason the application will include a module for the database, which in our case in Cloud Firestore. The module will be able to handle any oncoming data. The module will upload the converted data to the cloud, where it will give access to that data to whoever uploaded it. If anyone else needs access to the data, the module will be able to give other users access to it. The module will also handle retrieving any previously uploaded data.

4.3.2 | UML Diagram


Access Database
+ convertedData: arr
+ currentUser: str
+ listOfUsers: list
+ dataKey: str
+ uploadData(convertedData, currentUser): bool
+ giveAccess(currentUser, lostOfUsers): bool
+ checkAccess(currentUser): None
+ retrieveData(dataKey): arr

4.3.3 | Public Interface

All of the methods that are required to communicate with the database will be under the AccessDatabase class.

The uploadData method is responsible for taking the converted data and uploading it to the database. It will take in the converted data and the current user as the parameters of the method. Since the application will be connected to the database at all times, there is no need to pass in any connection parameters.

Once inside of the uploadData method, the module will access the upload function that is provided by Firebase. Before uploading any data, the module will create a collection for the data file, this will be used later on in the giveAccess method and checkAccess method. The data that has been passed in is split into two columns. The module will access the first column in the database and take the first column of the data and upload it. After that is done it will do the same with the second column of data. After all the data is uploaded, the module will access the giveAccess method to set the permissions for the data. After that is all done, the class will determine if the upload was successful or not. This boolean value is what the method will return so it can be displayed to the user.

The giveAccess method is responsible for giving the proper access to the data that is uploaded to the database. For the parameters it will take in the current user and a list of users that the current user can pass in. The current user and the list of users are who will be given access to the data in the database.

In the Cloud Firestore there will be a collection, or a list of users, for each data file uploaded. The module will access this collection and add the current user and list of users that was passed in. This will give those users access to the data if they are signed in with that account. The method will return a boolean value that says if giving access was successful or not.

The checkAccess method is responsible for checking if the current user that is signed in, has permission to access a specific data file that is on the database. The method will take in the current user as the parameter.

After getting the current user that is signed in, the method will check all of the collections, or list of users, that are in the database and find the ones that the user is in. The method will then take the list of data files that the user has access to make that data accessible for the user. The method does not have to return anything.

The retrieveData method is responsible for retrieving data that the user is requesting. Since the checkAccess method will have already ran, there is no need to check if the user has access to that data. The parameters for this method will be the id, or key, that is linked to the data file the user wants to retrieve.

Once in the method, the module will access the retrieve function that is provided by Firebase. For this function, it will need to pass in the id that was given to the method. After finding the data, the method will return the data retrieved so it can be used elsewhere in the application.

5 Implementation Plan

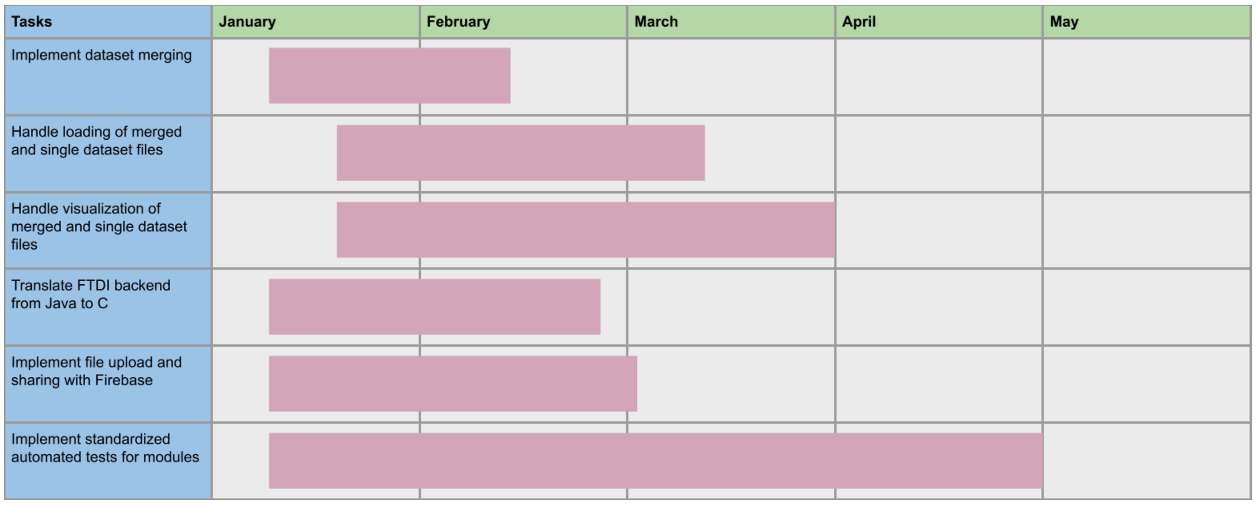
By the end of the semester, the team must have implemented an application which can achieve the following: (1) read data from a TOMST dendrometer and write the data to disk; (2) parse one or many datasets and create a graphical representation; (3) allow the user to manipulate the visualized data, and save the new dataset; (4) upload datasets to a Google Drive using the Firebase platform.

Reading data from a dendrometer has already been achieved; however, this process takes considerable time and utilizes an unofficial Java library to communicate with the FTDI chip which resides inside the dendrometer. Zachariah has taken up the task of rewriting the Java library in C utilizing the official FTDI API to communicate with the dendrometer.

Our application is already capable of visualizing one or more datasets for the user. However, the application does not allow for the merging of one or more datasets; in addition, laying graphs over one another is not properly handled. Given the duality of this situation, Asa and Nile have taken it upon themselves to work in tandem to implement these requirements. Specifically, Asa will work on merging two or more datasets, and Nile will work on visualization of merged datasets along with dataset manipulation. There is a point where the two will have to work together to define how datasets will be separated if stored in a single, merged file, however. This should happen early in the development process.

To give the application the ability to interface with Google Drive, the team has agreed to use Firebase – as per TOMST's recommendation. Niklas has taken up the task to write the module which will handle communicating with Google Drive through Firebase to transfer datasets.

5.1 | Implementation Plan (Gantt)



6 Conclusion

Our clients are currently using an application to collect data from dendrometers. However, when using this application they experience a lot of problems. The current application is only available on a windows laptop, meaning they have to carry a big heavy laptop up the tree with them. This can pose a danger to the equipment and the person involved. Another major issue our clients face is that sharing data they collect can take up 8 hours. They currently would have to upload the data to google drive, then the person who wants to access the data would need to download it. This is a very inefficient system, especially when considering we are dealing with a large amount of data. This team was formed to create a mobile version of the current application. Along with creating a mobile application, the team is implementing multiple upgrades to certain features. Making the whole process of collecting data and sharing data much easier and safer.

This document outlines the architecture that the team plans to follow when implementing the project. The document went over the architectural pattern the team is using for the project, which is the model-view-controller architecture. Allowing the team to split up the project into several distinct layers. The document also went more in depth for each layer explaining the purpose and functionality of them. A big discussion point in the document was providing a more detailed explanation of each of the project modules. There are three main modules that are involved in the project: CSV Module, Display Data Module, and the Database Module. In each module description you will find the description for the module, a UML diagram of the classes and functions, and a description of each component within the module. At the end of the document, the implementation plan for the project is provided. Giving a more detailed explanation on how the team plans to implement the major functionalities of the project.

As a team, we are pretty confident and happy with the current progress of the project. We have made good strides in the implementation of the functionalities of the application. With this document done, we feel pretty good about the overall state of the project. We built a pretty good and reliable architecture and design plan that we are confident in following. This document helps give the team a better understanding on what exactly we want to implement in the project.