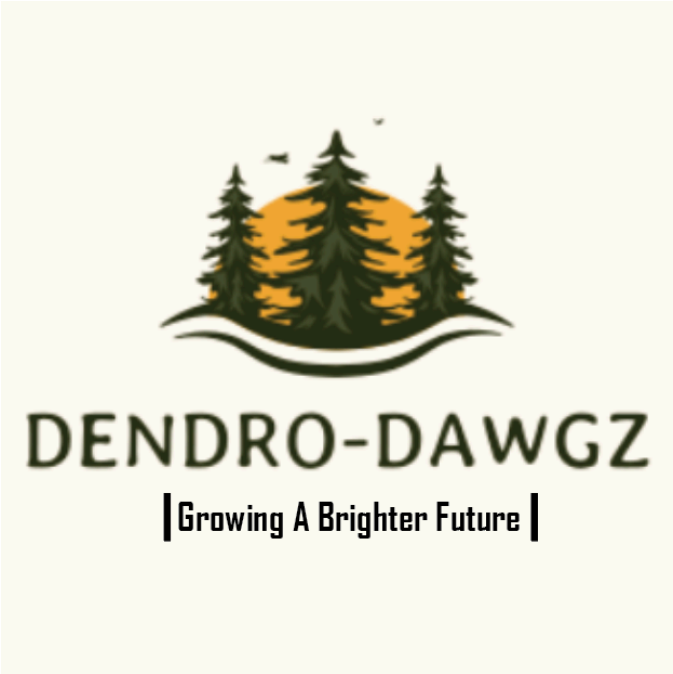
- Software Testing Plan -

Version 1
April 5, 2024



- 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

Table of Contents 2
Introduction 3
Unit Testing 4
Integration Testing 8
Usability Testing 11
Conclusion 16

Introduction

The DendroDawgz team is devoted to easing the process of obtaining and analyzing dendrometer data in order to support research projects around the world. Our long-term goal is to increase the overall flow of dendrometer data, leading to an improvement of the contributions of tree research. We have successfully created an alpha version of our application DendroDoggie with the ability to accurately perform all of our core modules. These include reading in data from the dendrometer, merging csv files, visualizing single and merged data files, and exporting data to the cloud. Before a final version release, we must run these processes through a series of tests to confirm their full functionality. There are three different types of tests to be performed on our application: Unit, Integration, and Usability testing. Unit testing relates to the actual testing of code and its outputs. We plan to run unit tests with JUnit on the most critical java files in our system. Integration testing involves ensuring that data is seamlessly transported throughout the application. This type of testing is important for our application because of its prevalence when switching between our application's multiple views. Lastly we have usability testing, which will be very useful for improving our user interface and user-friendliness. This process will involve having our clients complete a series of basic tasks inside the application. We are mainly looking to obtain feedback about the intuitiveness of our front-end, so that we can improve upon features of our app that may be difficult for new users to understand or reenact. The structured approaches described will ultimately lead to a more robust and user-friendly application. In the following sections, we delve into each testing area in detail, outlining specific test cases, methodologies, and expected outcomes.

Unit Testing

Unit testing is a strategy which picks crucial functionality in a software system to test. The tests can be run any time, but are generally run before code makes its way to end users. Unit tests are written to provide a benchmark for the system; this is useful when fixing bugs and adding new features because pieces of the software can be changed which could affect other pieces. By having a collection of tests to verify the software is producing expected results and to verify the software can handle edge cases, the integrity of a system can be confirmed during and after the development process. Additionally, unit testing can be a critical part of regression testing, allowing developers to quickly see if new code changes cause bugs or failed unit tests inside other components. This allows these developers to easily see if their changes alter previous fixes or versions of the code, and update the changes accordingly.

To ensure the integrity of the DendroDoggie application, the team has decided to employ the standard and widely used “JUnit” testing framework to test our software. Junit is the most commonly used Java testing framework on GitHub, and is consistently worked on and updated to provide the most testing tools possible. JUnit also can be used extremely easily within the Android Studio build process, by integrating the library with the gradle file. You are able to customize the test implementation runner within the build.gradle file, and simply adding JUnit there ensures that testing is properly integrated with your development process.

JUnit contains all of the expected test cases, including an initial setup of variables and environment, assertions, mocking and stubbing, test runners for suites and classes, and teardown - to name a few important functions. With these functionalities we will be able to thoroughly test the most important pieces of our software to ensure that everything works as expected, and continues to work as expected as future developers maintain it.

Over the following section, we will describe what classes we are testing and why, and go into specifics of testing inputs for the functions we want to have coverage of, setup requirements, and potential edge cases.

- pars.java
    - Because this class is used to parse the information that we receive from the dendrometer and other TOMST devices, this class is a critical piece of the core software that needs testing.
    - copyInt: inputs should be a string with an integer the length of two characters inside it at a given start point. Example: “12345678” starting at 0 with a count of 2 should return 12.
    - copyIntGTM: should be capable of handling the same input as copyInt, but additionally should handle hex values, and edge cases for negative UTC offset values. Anything over 0x80 should be subtracted from 0x80 and negated. For

example, “A8” should return -28, because 0xA8 - 0x80 = 40, and 40 to decimal is 28.
    - copyHex: Similar to the above functions, if the string is “A8” this one should return 168 as expected in hex.
    - dissasembleDate: This function uses the previous functions inside of it, and needs a class to be set up and initialized before testing. This should be tested with edge cases for the UTC offset, and normally expected values.
    - disassembleData: A class also needs to be set up and initialized for this function, and the inputs should contain both “ADC” and no “ADC” inside the input string for full coverage of the function. An assertion of all individual class values populated from the return will be run.
- CSVFile.java
    - The CSVFile class resides in the application’s backend, and provides a simplified interface for changing the contents of files, as well as adding, removing, copying, and transforming files for the programmer
    - toParallel: should accept a merged file which is in the “serial” format - where dendrometer data is listed one after the other - and restructure the data to be in “parallel” - where data for each dendrometer is separated into columns (i.e. there are N points of data on one line - where N refers to the number of dendrometers in the file for which data exists).
    - toSerial: should accept a merged file which is in the aforementioned “parallel” format and restructure the data to be in the aforementioned “serial” format
- GraphFragment.java
    - The GraphFragment class is responsible for implementing the functionality the users need to visualize and manipulate data. This class is a critical component of the user interface, and is the main piece with which the user will interact.
    - DisplayData: this function does not accept any parameter; however, the function works with data previously constructed by the *loadCSVFile* function. This function must be able to parse N collections of data from N dendrometers - where N communicates the number of data sets from a dendrometer contained within a file. More specifically, this function needs to be able to identify and extract each piece of data - differentiating between data from temperature sensors, humidity sensor, growth sensors, etc. - and organize the data into a structure which the chart can use to visualize data for the user. Testing this would entail processing some mock data, and then asserting the pieces of the charting structure are what is expected.
    - loadCSVFile: should accept a file path, and be able to load the contained data into memory for further processing and/or visualization. The function should be able to handle files with a single data set and merged files. The function should be able to keep track of which data belongs to which dendrometer. Testing this function

would require asserting the filled data structures have the expected data and the expected organization.
    - processLine: should be able to parse a line read from a data set file. The function should be able to identify if the line contains a serial number; otherwise, the function should be able to extract temperature, humidity, and growth data the device collected. The function should be device agnostic which would allow for our clients, and other users by extension, to visualize data from any of TOMST's devices.
    - mergeCSVFile: should accept an array of file names and compile the data set(s) from each file into one file - known as a merged file. The function must apply a header which contains the number of data sets, the serial numbers of the dendrometers' data contained within, as well as each dendrometer's latitude and longitudinal position. The function should also be able to merge data from a merged file with other, single data set files, and other, merged files. An example: if the user already has a merged file with 3 data sets in it, and wants to merge it with a file with a single data set, then the resulting file would count "4" data sets, list the 4 dendrometers, as well as contain the data for each dendrometer present in the source files. If the user has two merged files - one has 3 data sets, and the other has 4 data sets - the resulting file would count "7" data sets, and contain the data present in the files which were merged.
    - LoadDmdData: does not accept any parameters; however, the functionality through which the user can visualize data as they collect data from the dendrometer. Therefore, it is imperative the function can parse data - coming from the dendrometer - and put the data somewhere from which it can be changed and saved. So, the function needs to be able to access and read data from the dendrometer - passed through the DmdViewModel by the HomeFragment - and parse the data, then collect the data into a construct which can be visualized, and finally convert the construct into an object which the chart can use to visualize.
- ListFragment.java:
    - This class is responsible for updating the list that resides in the File Viewer page of the application. Some of the functionality within this class is responsible for reading and writing the csv files to the cloud. The class also handles the initialization of merging csv files into one csv file. In order to initially populate the list the class will pull files from the device and the cloud.
    - loadAllFiles: This function will take all the files that are stored on the device and populate the list with them. The function should be able to take the list of files on the device, fFriends, and create an adapter with it. Then it will take that adapter and set it for the list view. In order to test this we would need to check if fFriends has the correct csv files. We would test this against the known list of csv files on the device.

- loadFromStorage: This function should be able to take all the necessary csv files from the cloud and load them into the list view. It will do this by going through each csv file on the cloud. Then for each file it will check if the current user is in the list of users that has access to the file. The list of users with access is stored in the metadata of the file. If the current user has access to the csv file, it will download the file by calling the downloadCSVFile function. This will add the file to fFriends and the function will then update the list view with this new fFriends by setting a new adapter for the list view. In order to test this functionality we will check if the current user is actually part of the user access list. The test will also include checking if the pulled file from the cloud is in the fFriends list.
- downloadCSVFile: This is a fairly simple function that will take in the file name and file path. It will then pull the file from the cloud based on the file name input. Then for this file it will download it to the path that got inputted. To test this function all we have to do is check if the file is in fFriends.
- uploadDataToStorage: This function is responsible for uploading csv files to the cloud. The function will take all the files that are selected and go through each one. To get the selected files it will check the selected flag for each file in fFriends. For each file, the function will upload it to the Files folder in the cloud. After a successful upload, the function will update the metadata to include the user who uploaded the file in the file access list. There are two things to test here, if the file got uploaded and if the metadata got updated. To test if the file got uploaded, we would go through the files in the cloud, and check if the file is in there. Then if we do find the file, check if the metadata includes the user who uploaded the file.
- updateMetadata: This function gets called from the shareData function. If a user clicks share and inputs email addresses, the shareData function will then call updateMetadata. The updateMetadata function is responsible for adding users to the file access list. The function will take in an array of emails and go through each selected csv file. Then for each file, the function will go through each inputted email and that email to the user list for the file. This allows users to share files with each other. In order to test this, we would take the file that got updated and check if all the emails got properly added to the access list.

Integration Testing

Integration testing is another crucial part of ensuring software works as expected and even fulfills initially given software requirements. Unlike unit testing, this version of testing works by comparing values that are passed between interfaces instead of focusing on the specific assertions of function return values.

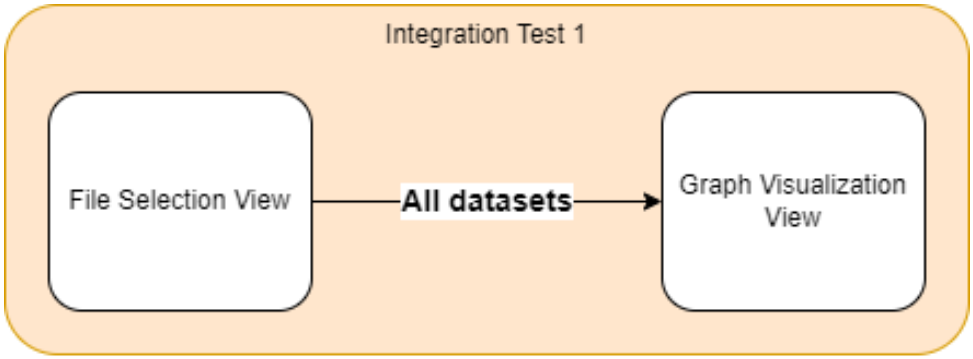
For example, if within one view of an Android application, you input your user credentials and it takes you to a separate view with that user's data, you want to ensure that the credentials properly passed from one view to the other, and that the Android application went to the intended view after checking the user's input.

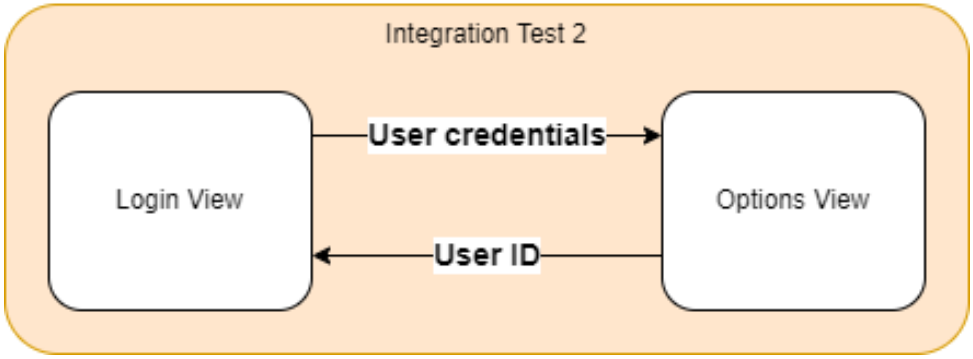
Integration testing in our Android application will be fairly straightforward, as we only have a few interfaces that we need to test proper flow and data passing. For our tests, we are going to use an emulated environment present on Android Studio, and use the robust logging and debug system to ensure that all components are thoroughly examined and meet our defined requirements.

- For our first integration test, we will be analyzing the relationship between the file selection view and the graph visualization view, and the process of passing large amounts of data between these views. It is important that the data is the same between views, and that the application is capable of handling several use cases and functional requirement metrics.
    - Preparing the test data: First our emulator needs to have the required test datasets present on the device. There will be four datasets that are examined: one large dataset with 50,000 entries and a header, one small dataset with 1,000 entries and a header, one dataset with 1,000 entries with no header/metadata information, and two datasets that will be merged during the selection/viewing process: the 50,000 and 1,000 datasets will be merged together with the appropriate header information to distinguish them. The first three datasets need to be created and placed in the emulator's internal storage. The fourth will be created during runtime.
    - Setting up metrics: For this test we will be using the logging system and appropriate tags inside Android Studio to ensure metrics are met manually. For this integration test we need to be certain that serial numbers given by the datasets are properly passed, the actual data is properly passed, and the time taken for loading/passing each dataset is measured. To do this, serial numbers will be logged in the graph visualization view, the 100th (40,000th for the large sets) data points will be logged in the graph visualization view, and the time of execution for both views will be logged. Comparisons will be made afterwards.

    - Executing the tests and monitoring the results: Inside the emulator - after setting up the environment - the tester will select the datasets and execute them in the order provided above. First the user will test the 1,000 entry dataset and monitor the logs for the given metrics, then the 50,000 entry dataset, then the 1,000 entry dataset with no header, and finally the user will select the 1,000 and 50,000 datasets, which will be merged during runtime.
    - Reporting and analyzing results: The serial numbers in the graph visualization view will be compared to the CSV file's serial number to see that they are equal, the selected data points will be compared in the graph visualization view against the CSV file's data points to see that they are equal, and the difference between times between views will be compared against a given value. If these values are all correct, then this integration test is good to be signed off.

- For this integration test, we will analyze the flow between log in view and the options view. We will look at if the user data is getting properly passed between the two views. It is important that the correct user information is getting passed as we don't want to distribute the incorrect information. Also important to make sure that data is actually being passed between the two views.
    - Preparing the test data: For the data we will use three different users. Each user will have a different email and password. There is no data that needs to be added to the emulator itself.
    - Setting up metrics: For the actual test, we will use the logging functionality inside of Android Studio. For this test, we need to make sure that the user id that logs in is the same user id that gets passed to the options view. When user information is inputted into the login form, we will log the user id associated with that user. Then when the user logs in, we will then log the user id that was logged in. The final log will happen when the user enters the options view. Here we will log the user id that is being read.
    - Executing the tests and monitoring the results: After everything is set up, inside of the emulator the tester will be able to start the test. First the tester will take login info for user 1, and input them into the login form. At this point, the tester should be able to see the user id displayed in the log. After clicking login, once again the tester will look at the log and see the user id displayed. Then the tester will move to the options page and look at the log and see the user id displayed. The tester will repeat these steps for the other two users.
    - Reporting and analyzing results: The three different user id's displayed during the test will be compared to the known user id for each of the three users. If the user id's match, then we can confirm that the integration between the two views is working correctly. If there is any discrepancy, then we can further test to find the problem.


Integration Test 1
File Selection View
All datasets
Graph Visualization View


Integration Test 2
Login View
User credentials
User ID
Options View

Usability Testing

Usability testing is the final piece of testing that a good piece of software should implement. This testing ensures that the actual end product meets the usability requirements set out before it during the requirements acquisition phase, and that the software is actually usable. If the application is extremely non-intuitive, then no matter how functional the application actually is, users will have no idea how to access that functionality. This is why usability testing is so helpful: it helps the development team see exactly what where users struggle with the interface of the application.

For our Android application, the usability testing that we will implement is user testing. We luckily have access to the four main users of our application, and so we are able to watch them perform several steps of the applications workflow, and precisely see where they falter or where the interface is unclear. We can then use this information to make our UI/UX better and more tuned to our clients' thought processes. The four users of our application are members of the NAU ECOSS, and prominent researchers in the environmental science field, and will be the main users of this application. They are Andrew Richardson, Mariah Carbone, George Koch, and Austin Simonpietri.

- The workflows that we want to test with the users are the most critical pieces of our software. We only have a few key workflows and processes in our application, and so it is critical to us that these are as intuitive as possible. The following workflows are the workflows we will be testing with our users:
    - Data reading: downloading data is a key workflow for our Android application, and therefore needs to be thoroughly vetted before releasing to production. This workflow from a top-level point of view goes as follows: the user may select options to dial in the downloading, then the user plugs in a TMD adapter, approves permissions, and plugs the device in until it is downloaded. For each of these pieces of the workflow, the application must be very usable and intuitive, or else users are not capable of comfortably using the most important function of the application. Therefore, our application will ensure that following tests and requirements are followed and met during the usability test:
        - Step one: user selects download options:
            - The user should be able to clearly see where to navigate within the application to find the download options. The options themselves must be short, clear, and self-explanatory for a typical user of the application. The following are selections from worst to best for finding the options menu:
                - “I could not find where the button was located on the screen”

        - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
        - "I found where the button was, but the icon could better communicate"
        - "I was able to navigate to the options screen"
    - The following are selections from worst to best for selecting the options:
        - "I did not know what any of the options were for"
        - "I was able to intuit what a few of the options were for"
        - "Most, if not all, options were clear and well-described"
- Step two: downloading data
    - The user should be able to navigate back to the home view from the options menu, and that will be judged by the same navigation test above. The application should clearly state next steps for downloading, and accurately describe the permissions required for the app. The application should additionally present useful information when downloading data. The following selections from worst to best will determine results of this section's usability:
        - "I was unable to determine what to do next to download data"
        - "It was clear to me what the next step would be to download data"
    - The following selections from worst to best will determine results of this section's usability:
        - "No useful data was presented to me during the download"
        - "There was some data presented during the download, but not all of it was useful, or some was missing"
        - "All of the data presented during the download was useful, and none was missing"
- Viewing data: The visualization of data is the most critical module of our application. This process must be completed with extreme precision to ensure that users are obtaining data that is completely consistent with the original dendrometer readings. The graphs created from our charting library should be navigable, simple, and efficient. Efficiency of a graph relates to the productivity and convenience it supplies to a user. Users should easily obtain valuable data quickly proceeding the rendering of the graph. This process is very simple and only requires 3 'clicks'. The following steps will be performed for usability testing of the visualization of data in our application:
    - Precondition: User has CSV file stored on device in documents folder

        - Step one: user navigates to ‘file viewer’ tab
            - The user should be able to intuit which button in the navigation menu will take them to the aforementioned page; it is crucial the icon communicate this; therefore, from worst to best, the following statements outline the experience for finding the button:
                - “I could not find where the button was located on the screen”
                - “I know to use the navigational menu, but the button was hard or indistinguishable from other buttons”
                - “I found where the button was, but the icon could better communicate”
                - “I was able to navigate to the file selection screen”
        - Step two: User selects a csv file to visualize
            - The user should have the ability to easily select/deselect any of the files. The following statements list possible experiences for selecting/deselecting datasets:
                - “I could not find my csv files in the application”
                - “I am stuck on how to select a file”
                - “I am able to select a file, but cannot deselect it”
                - “Files are easily selected and deselected”
        - Step three: user finalizes selection, and device visualizes the data.
            - The user should be able to go straight to the button to finalize their selection and display the csv file; it is important the button be obvious and communicate its function for new and returning users; therefore, from worst to best, the following statements outline the experience for finalizing file selection:
                - “I could not find the button to finalize selection”
                - “I found the button to finalize my selection, but it took some time to figure out which button performed this function”
                - “I knew pretty much instantly which button to press to perform the visualization”
    - Merging data: merging data sets should be a straightforward process which only requires the user to follow two or three steps with each step only taking a few seconds to a minute. By definition, this process is intuitive for the user, and allows them to get to visualizing and reviewing data as quickly as possible. Therefore, when testing, the user should be able to complete this step in 2 minutes - allowing for time to choose data sets; there user should be able to intuit how to select data sets to merge, and be able to find the button to visualize with ease. Therefore, the

following steps will outline the tests which need to be met, as well as the degrees for success to communicate which aspects of the process must be improved.

- Step one: user moves to the file selection page
  - The user should be able to intuit which button in the navigation menu will take them to the aforementioned page; it is crucial the icon communicate this; therefore, from worst to best, the following statements outline the experience for finding the button:
    - "I could not find where the button was located on the screen"
    - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
    - "I found where the button was, but the icon could better communicate"
    - "I was able to navigate to the file selection screen"
- Step two: user selected a number of files to merge
  - The user should be able to begin the process of selecting files; it is very important the user understand which files will be merged and which files will not be merged; therefore, from worst to best, the following statements outline the experience for selecting files:
    - "I got stuck trying to select files to merge"
    - "I could select two or more files, but I need an indicator of which files will be merged"
    - "I was able to select and deselect the files I wanted and did not want to merge"
- Step three: user finalizes selection, and the phone is able to visualize the data
  - The user should be able to go straight to the button to finalize their selection and merge the files; it is important the button be obvious and communicate its function for new and returning users; therefore, from worst to best, the following statements outline the experience for finalizing file selection:
    - "I could not find the button to finalize and merge the selected files"
    - "I found the button to finalize and merge, but it took some time to figure out which button performed this function"
    - "I was able to find the button to finalize and merge the selected files, and visualize the data"

- Exporting data to the cloud: Being able to export data to the cloud is a key piece of the workflow. This allows users to share data easily across any device. This should be a very simple and quick process. The high-level overview of this would

first be the user navigating the file viewer page. Then the user should be able to select any amount of files that they want to upload to the cloud. Once selected, the user should be able to press the upload button, allowing the files to be uploaded to the cloud. The following tests will be performed to complete the usability testing for exporting data to the cloud.

- Step one: user navigates to file viewer page
  - The user should be able to tell what button on the bottom navigation will lead to the file viewer page. The following statements outline the experience in navigation to the file viewer page:
    - "I could not find where the button was located on the screen"
    - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
    - "I found where the button was, but the icon could better communicate"
    - "I was able to navigate to the file viewer page"
- Step two: user selects file(s)
  - The user should be able to tell how to select files. They should also be aware that they are choosing files to be uploaded to the cloud. The following statements outline the experience in selecting files to upload:
    - "I got stuck trying to select files to upload"
    - "I could select one or more files, but I need an indicator of which files will be uploaded"
    - "I was able to select and deselect the files I wanted in order to upload them"
- Step three: user presses upload button and files get uploaded
  - The user should easily be able to recognize how to upload the selected files. The following statements outline the experience in selecting files to upload:
    - "I was not able to figure out how to upload selected files"
    - "I was able to find the upload to cloud button but could not upload the selected files"
    - "I was able to find the upload to cloud button and upload the selected files"

Timeline for Usability Testing:

- All four usability tests will be done at the same time, during one of our weekly meetings towards the end of April with the four main users of our application: Andrew Richardson, Mariah Carbone, George Koch, and Austin Simonpietri.

Conclusion

In conclusion, the testing efforts outlined for the DendroDoggie application have been planned and designed to ensure reliability, functionality, and user satisfaction. Through a combination of Unit, Integration, and Usability Testing, we cover all critical aspects of the application's purpose.Unit Testing involves testing individual units or components of the software to validate their functionality. We will do this by employing the widely-used JUnit testing framework. We aim to thoroughly test critical components such as the pars.java, CSVFile.java, GraphFragment.java, and ListFragment.java files. Specific test cases are designed to cover various scenarios and edge cases to ensure robustness and reliability. Integration Testing focuses on testing the interaction and data flow between different components or modules of the application. Specifically, we will analyze critical workflows, such as file selection and graph visualization, to ensure seamless user experience and data integrity. Usability Testing serves to assess the intuitiveness of the application's interface. By observing the feedback from real users interacting with the application, we identify necessary areas of improvement. We plan to instruct our clients to complete basic application tasks such as data reading, viewing, merging, and exporting to the cloud. Given our application’s role in providing precise and critical environmental data, rigorous testing is an essential process to complete before releasing our application to the public. DendroDawgz continuously aims to supply its users with high-quality and reliable software.