CA400 Technical Guide

Project Title: LiveFocus
Students: Cormac Duggan and Sean Hammond
Student Numbers: 17100348, 17374356
Supervisor: Graham Healy
Completion Date: 7/5/2021

Abstract

In the current environment of online work and school in the form of synchronous and asynchronous lectures, it can be hard to retain a student's focus over an extended period of time. With the extent of information and distractions available to students at an instant, it is almost a given that unengaging learning environments will lose the attention of the people they are being provided to. To combat this inevitability, we have developed an application which will allow lecturers or video creators to track their viewer's attention retention. With the use of a Muse EEG device, users will be able to record and analyse the results of a viewing session of one of their provided videos. With sufficient sample sizes they can then find key points in a video where the viewer's attention is lost and adjust their videos accordingly prior to distributing them to students.

Table of Contents

Table of Contents

1. Introduction 3

a. Overview 3

b. Motivation 3

2. Research 4

a. Application Implementation 4

b. Signal Processing 5

3. Design. 6

a. System Architecture Diagram. 6

b. Context Diagram. 7

c. Use Case Diagram. 7

4. Implementation 8

a. User Interface 8

b. Muse Connection 9

c. Processing 9

d. File Manipulation 10

e. RESTful API 10

f. Database 10

5. Sample Code 11

a. Creating the UI 11

b. Adding Action Listeners 11

c. Graphing Data 12

d. Calculations 13

e. Muse Interactions 13

f. API 14

6. Issues Resolved 15

a. UI Design 15

b. Active UI Implementation 15

c. Muse Device Connection 16

7. Results 17

a. Results 17

8. Future Work 18

a. Development 18

b. Research 18

9. Sources 19

a. Dependencies 19

b. References 19

1. Introduction

1.A Overview

This document reviews the technical aspects of LiveFocus, which is our application that attempts to measure, record, and provide a platform for analyzing EOG and other neural activity signals. The research, design, implementation and samples of the code, issues, outcomes and future work possibilities will all be discussed here.

Using bluetooth connectivity with a Muse headset, our platform allows users to track EOG and accelerometer data while watching a video. They then have the ability to analyze the data themselves, or upload their recordings to a server and allow others to analyze their recordings instead. When viewing recording data, any number of different recording files can be used at one time allowing users to look for correlations between different people that have watched the same videos. The purpose for this would be if someone creates a video and wants to analyze users' neural activity while watching it. They could send the video to multiple people who upload their recordings to the server as feedback for the video maker. The video producer could then collect the feedback recordings and analyze them with respect to their video.

While we do not have the expertise to create a system that can analyze the readings and identify event artifacts from the recording data, we did find papers supporting the concept. In the research section of this document we discuss our findings and different papers that prove in time there could be correlations between other work being done in neural computing and this project.

1.B Motivation

Our motivation for developing this application comes from the difficulty experienced by students in the context of online learning, and the challenges that lecturers face when creating video lectures and guides. Online learning has become the primary method of education for higher level institutions in the past year due to the rise of COVID-19. Because of this, we felt that developing software that could provide insights into creating online educational media that can effectively hold the attention of students while also allowing them to engage in course material would be an extremely beneficial system to build. As Final Year students ourselves who have experienced the issues of poorly made video lectures first hand, providing a possible solution to these problems was something that greatly motivated us to pursue this project.

2. Research

2.A Application Implementation

When implementing the application, the first and most important thing to do was to find a way to connect to the muse device and record the signals that it created. Luckily there was an API designed to actively record data from the device over bluetooth that we could use to our advantage. After installing it and testing it out to ensure it works we read through the documentation to see what the returned data meant as at first it was unclear. However, as a command line program we found it easy enough to implement a way to connect to the device and record data once we understood it all.

As we had decided when planning our project, we implemented our application almost exclusively in Java, making this our first attempt at using java swing. To get a grasp of how everything worked, we read through the documentation first for all the different types of features that could be added to an interface. After that, we proceeded to tinker around with some basic interfaces to get used to how to implement them properly on a larger scale.

Following that, as it wasn't provided in java swing, we needed a library for displaying graphs and videos. JavaFX had the ability to display videos in a swing UI just as we needed and FFMPEG would allow us to clip videos further down the line in the project. XChart was the jar we settled on for graphing data in the UI as it provided all the minimum functionality we needed to create the charts we wanted. Reading through the documentation for XChart we also found a list of features that we wanted to implement. Although, later on in the project we found that the UI would crash if we implemented the majority of XChart's additional features and thus left them out.

To create the database we used MongoDB as it is a NoSQL database and is really straightforward to use. To connect to our database we found that a RESTful API on the redbrick pygmalion server would be most beneficial, as we could use it to store files, retrieve files, and store file pointer data as well as user information in the database.

After some implementation we found that a large amount of our class's functions would have a void return making testing with JUnit difficult. We decided to use Mockito in combination with JUnit to do all of our unit testing for the application. Mockito would allow us to create mock instances of classes and check that specific lines of code are being run correctly. This was a good workaround for testing any of the many functions that didn't return anything after they finished.

2.B Signal Processing

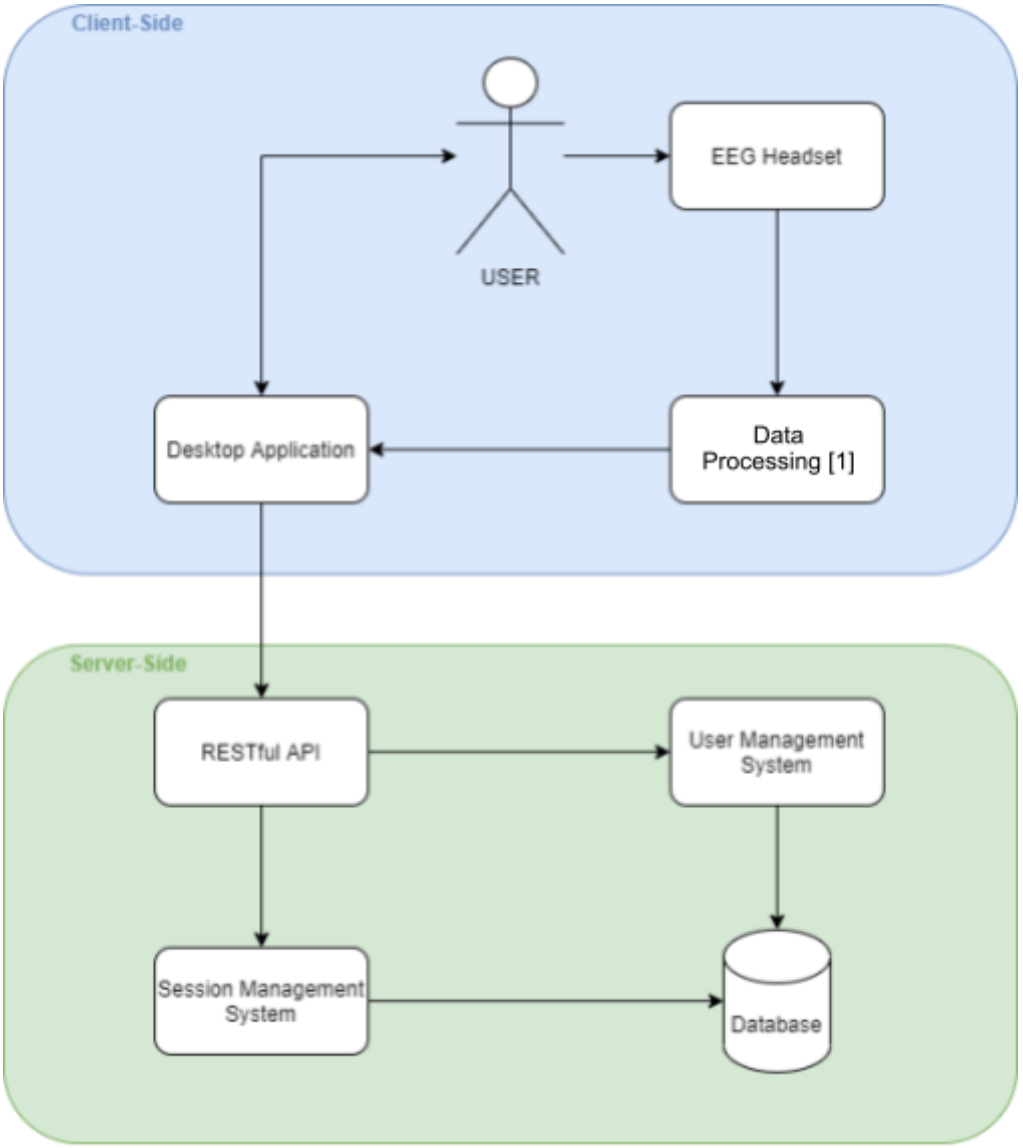
In the past decade, research in neural computing has seen a large rise in interest and development. In particular research into the analysis and usage of signals recorded using electroencephalography and electrooculography has seen steady, if not increased progress, especially in the past year. Many research documents can be found pertaining to signal processing and brain computer interfaces using the processed signals. While most of the research isn't directly correlated to how well someone is paying attention to something, progress with these forms of signal processing only serve to improve the purpose of LiveFocus.

The first of the papers we looked at which had the best correlation with our project utilized CWT (continuous wavelet transform) to extract artifacts of EEG and EOG signals in which both alpha signals produced correlated changes. From these extracted artifacts, an LSTM (Long-Short Term Memory) neural network is used to classify whether or not someone has become more or less drowsy over the allotted period of time.[1] The most important thing we noted from this document was that, while the signal processing itself may be out of our area of expertise, future research could provide the insights we are looking for in our data. To support this idea we made sure to design the system so that any signal processing and artifact identification could be added later on with minimal change to the system as a whole. (See 3.A [1])

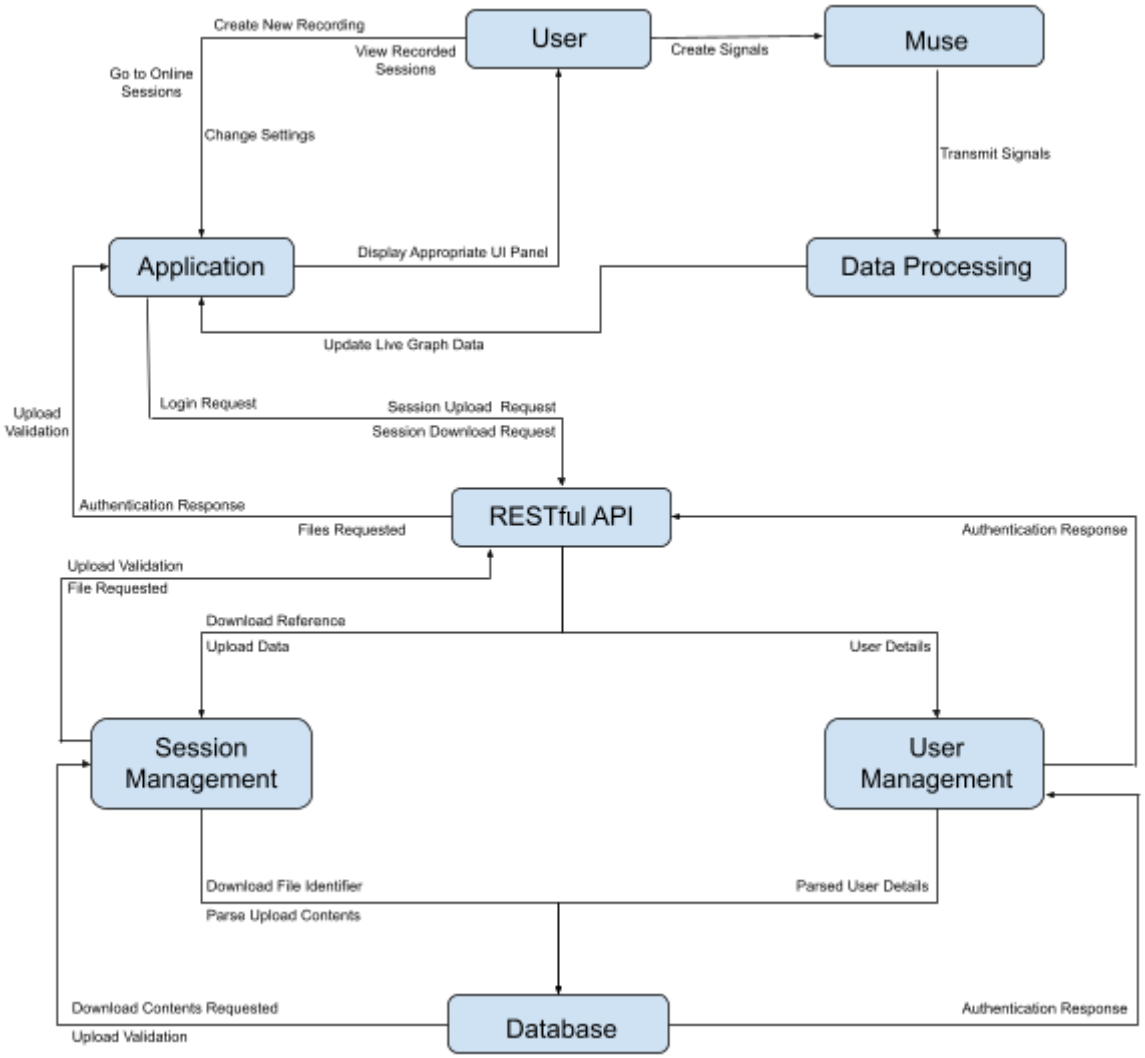
Other papers we looked at were less directly related to our topic, but still proved useful in one way or another and showed that research is being done on the same signals we are using here. One of these papers pointed out that EEG signals can be difficult to monitor for artifacts as they can be somewhat corrupted with EOG readings which need to be removed.[3] Because of this in our implementation we used EOG signal readings as our base line. We weren't sure how to implement the discussed $H\infty$ method of removing corruption from EEG and ensuring it worked properly, so it seemed safer to use EOG, as the corruption appeared to be one-way. Another of the papers we looked at discussed using EOG signals to determine if movement was occurring based on large amplitude fluctuations in the recordings.[2] However, as we found the Muse device was able to record accelerometer data, we found that would be easier to implement and more useful for our purposes anyway.

3. Design

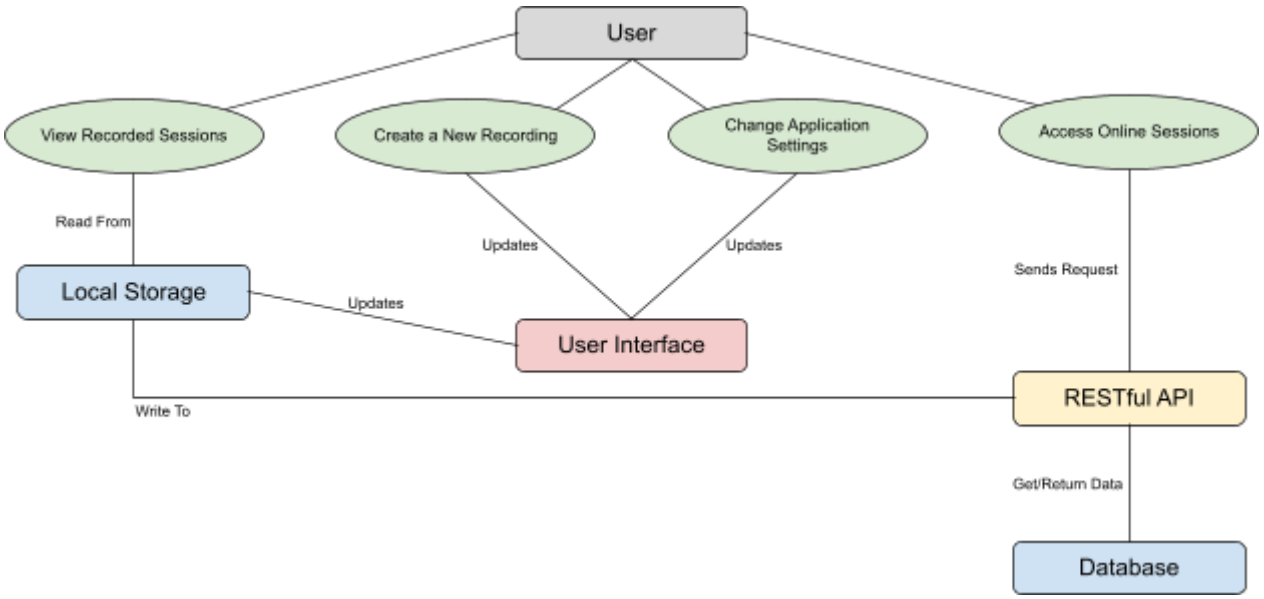
3.A System Architecture Diagram



3.B Context Diagram



3.C Use Case Diagram



4. Implementation

4.A User Interface

When developing LiveFocus, the very first thing that we started on was the user interface. We began to put together some sort of interactivity for users to see just where we would need to connect to the backend of the application. Using java swing for the first time we began with a naive approach and set the bounds of every object we created in the interface. Issues quickly arose with objects not appearing where we wanted them to as we had a lack of complete understanding of the Grid Layout that was being utilized. On top of things not appearing in the right places, our code also very quickly became extremely difficult to manage as there weren't comments, rhyme, or reason to how or to the order in which things needed to occur.

After a short time of trying to refactor our existing code into a usable state we decided to return to researching more about different ways to implement swing interfaces. We came across the Form Layout in the jgoodies jar, and once fully understanding how it worked we decided it would be far more beneficial to us than the previous Grid Layout. The form layout would allow us to remove any placement of objects in the interface that were static (e.g. A button). The form layout uses form xml documents to define the location and properties of every piece of the interface. This meant that the hundreds of lines of java that were previously used to create, style, and place objects in the interface could be replaced with a single form. Knowing this we quickly rebuilt everything we had up to that point within 3 days and were able to keep future code far more legible and compact.

After deciding to use the form layout we encountered an issue where no objects could be edited on the interface unless it was an edit of changing class variables. For example, a JButton has an option to change the text on the button. This would be a legal change as the button text is a class variable. However, if we wanted to add a new item to a JPanel, as the children components of the JPanel are not class variables, the Form Layout would cause an error and hang the application. After some experimenting, we found that JScrollPanes would be the best solution for this issue. The JScrollPane has a class variable (view port) which can be any swing object. This meant that any time we wanted the interface to update reactively we would need to create the content in java and set the scroll pane's viewport to the content we wanted. On these scroll panes we would be able to add charts from the XChart jar to graph data.

Having all the knowledge needed, we proceeded to create the interface and all the corresponding action and change listeners for the components of the interface. While some of the interface classes would have some functionality, they were mostly just there as classes to initialize. The action and change listeners on the components of the interface were the important part of the code which connects the interface to application functionality.

4.B Muse Connections

Connecting to the Muse device was the second most important hurdle to overcome. Without the ability to connect to the device, any of the functionality in the interface would be of no use to anyone. After scouring the internet we managed to find the Muse API that allows connection via bluetooth and writes readings of the device to command line interface.

With the Muse API we were able to connect the device to our application with a simple command from java. With this the device would connect and be able to record data that would be parsed and written by other parts of the application. However, we found that if the process was called to run the device connection and recording the UI would hang until the process was killed. Because of this the Muse recording system required multithreading which is slightly different when using swing. In the case of a swing UI a second thread requires a SwingWorker class instance instead of a basic thread to allow the interface to continue working. With the SwingWorker implemented to allow device connection and recording we were able to move on to processing the data being collected.

4.C Processing

With the ability to connect to the device and actively record data from it, we needed to be able to parse the data returned by the muse API. We decided to create a helper class called Processing, where all data would be parsed from the device readings and sent back to the application to be displayed and written to files appropriately.

At first glance, it wasn't clear what the data meant as the muse readings would write over 100 lines per second with a string followed by 3 to 4 doubles. After reading more on the documentation for the recordings we found that the string was formatted in a way such that the type of recording was the last set of characters before the doubles appeared. Then the following values correspond to the node locations on the device or the single data type recorded. For example in a second we could get over 100 readings looking something like "xyz-abc/wxy/eeg 12.43 15.87 6.21 8.59". In this sample case the 12 refers to the eeg reading of the node on the muse located behind the right ear, 15 - the right forehead, 8 - the left ear, and 6 the left forehead.

We decided to restrict our measurements to the accelerometer, as well as front and rear eog signals. When data was being collected the Processing class would parse the data returned by the device and, if it was accelerometer data, parse the 3 doubles into a single change over time value. If the data collected was eog reading we would take the variance over a given time defined by the user prior to starting the recording.

On top of actively processing data, the Processing class was also used to calculate the averages for when a user went to view previously recorded sessions, as well as calculate the correlation coefficient of two readings.

4.D File Manipulation

When a user creates a new recording, a file is created using the details that were input in the file creation form. A group, video path, time of creation, and what signals are being recorded are entered in the first line of the file as metadata. If the file name that was input by the user already exists in the save directory, then that file will be overwritten with the new contents. Once the recording begins, every line following the header will represent 1 interval of data for every possible type of recorded data. For example, if we only wanted to record the accelerometer data the file would contain two zeroes, one each line for the unrecorded eog data and the accelerometer reading.

Once a file has been created, it can be uploaded to the server or viewed on the view sessions page of the interface. However, if the contents of the file have been edited after creation the file will be excluded from options to view. This is to ensure there are no crashes in the UI that cause the application to hang. Assuming all the contents of the file can be read properly, the data contained in it can then be displayed on a chart for the user to view and analyze.

4.E RESTful API

The RESTful API is run in Python using the Flask library, and allows connection between the user and the online functionality of LiveFocus. When a user chooses to login, the API will query the database for the user details entered by the request. If the database contains the correct details then an authentication will be passed back to the user and they will be logged in. From there, if they choose to upload a file the API will store the file on the Flask server and update the database with the file details and location of the file. If the user searches for a file the API will again query the database and return whatever exact matches exist in the database for the request. If the user then chooses to download a file the API will get the matching file location and respond with the requested file for the user to save to their local directory.

4.F Database

With the RESTful API that connects to the database, there wasn't a large amount of data that needed to be stored. As the files that can be uploaded and downloaded via the Online page of the interface are stored on the Flask server, the database is only needed to retain user details and file info and their locations. That way the API can query the database for a specific group, user, or file name and return all possible files to the user's application.

5. Sample Code

5.A Creating the UI

The first step when opening the application is to build a user interface that will be loaded. By inheriting the contents of the matching .form files, each of the pages throughout the application will have all the necessary information to build the base of their functionality. From there, the JFrame containing the interface must be displayed and appropriate action listeners must be added to the contents of the interface.

```
public Record(JFrame frame, int width, int height) {
    //create frame
    frame.setSize(width, height);
    frame.setContentPane(p1);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);

    //add action listeners to all link buttons on right side of ui
    new LinkButtonsInit(p1, frame, onlineButton, recordButton, sessionsButton, settingsButton);

    //add remaining necessary action listeners and tool tips
    newRecordingButton.addActionListener(new RecordNewRecordingButtonActionListener(frame));
    newRecordingButton.setToolTipText("Create a new recording.");
}
```

5.B Adding Action Listeners

In order to test each of the classes defined in the application properly, all the action listeners needed to be extracted from the class which utilized them and made to be their own class. As seen in 5.A when action listeners are added their classes follow the naming convention of [Parent Class Name, Item getting listener, “ActionListener”]. As almost no buttons in the application do the exact same thing, they would all need a related class. In this example the newRecoding button found in the GUI.Record class creates an instance of a RecordInfo class, which is the form where information about the file you would like to create is filled out.

```
public RecordNewRecordingButtonActionListener(JFrame parentFrame) {
    frame = parentFrame;
}

@Override
public void actionPerformed(ActionEvent e) {
    //create jFrame for new recording form
    JFrame formWindow = new JFrame();
    new RecordInfo(frame, formWindow,  width: 700,  height: 500);
}
```

5.C Graphing Data

Using the XChart jar, an XYChart is created which graphs a pair of double Arrays. One array being the Y values and one being the X values. It is very important that both Arrays are the same size otherwise the chart will produce errors. In the class definition and all other classes the chart is defined to be 100 data points along the X axis to retain a uniform size. To move the data that is being displayed a new pair of Arrays for the X and Y values are created. Then each of the sets of Y values displayed on the chart is updated with the new information. The series that represents the average is updated elsewhere.

```
//move existing graph multiple data points to the left, maximum 100 per movement. used for review sessions
public void moveCountLeft(double[][] items, String[] names, int displacement) {
    //adjust the times array based on how far it needs to move
    for (int i = 0; i < times.length; i++) {
        times[i] = times[i] - (displacement * .25);
    }
    time -= displacement * 0.25;
    //adjust the series data to match the new times
    for (int q = 0; q < items.length - 1; q++) {
        double[] tmp = new double[100];
        for (int i = 0; i < items[q].length; i++) {
            tmp[i] = items[q][i];
        }
        chart.updateXYSeries(names[q], times, tmp,  newErrorBarData: null);
    }
    //reset average to all zeroes. will be updated after all series have correct new values
    chart.updateXYSeries( seriesName: "Average", times, new double[100],  newErrorBarData: null);
    //redisplay new chart
    cp.repaint();
    cp.revalidate();
}
```

Styling of the chart is also contained in the Graph class as it needs access to the class attributes in order to update the display properly. Options like the following (toggleSeriesOff) allow JCheckBoxes to be used in order to display or hide each line that is contained within the chart.

```
//hide a line from view on the chart
public void toggleSeriesOff(String name) {
    XYSeries xySeries = chart.getSeriesMap().get(name);
    xySeries.setLineStyle(SeriesLines.NONE);
    //redisplay new chart
    cp.repaint();
    cp.revalidate();
}
```

5.D Calculations

There are a variety of mathematical calculations that are used to process and interpret the data readings as well as calculate variance, averages, and correlation coefficients. In this sample the function determines the variance of a list of doubles and returns the resulting double. All these relevant calculations are found in the Processing class.

```
//calculate the variance of a list
public static double CalcVariance(List<Double> readings){
    double total = 0;
    for(int i = 0; i < readings.size(); i++){
        total = total + readings.get(i);
    }
    double mean = total / readings.size();
    total = 0;
    for(int i = 0; i < readings.size(); i++) {
        total = total + Math.pow(readings.get(i) - mean,2);
    }
    return total/(readings.size()-1);
}
```

5.E Muse API

In order to obtain data from the Muse headband, two programs needed to be run asynchronously in the background in SwingWorker threads. These two programs were the Muse APIs for sending and receiving data, muse-io and muse-player. The thread that runs the muse player, the API for receiving data, reads each line of the program output. If the session has started recording and the appropriate amount of time has passed (the granularity) to ensure data is not being captured dozens of times per second, the output line is checked to determine if it contains the correct data type. If it does, the readings given in that line are processed and added to the graph, and stored in an Array List, to be written to a file.

```
while ((line = input.readLine()) != null) {
    dataTypeCurrent = dataTypesMap.get(dataTypes.get(dataTypeIndex));
    //If line that was read in contains the ID of the data type currently being captured, do processing
    if (line.contains("/" + dataTypeCurrent) && captureData) {
        //get time between now and last time data was taken in
        long finish = System.nanoTime();
        long elapsedTime = finish - start;
        double elapsedTimeSeconds = (double) elapsedTime / 1000000000;
        //Check if elapsed time is greater than relative granularity, if so, continue processing captured data
        if (elapsedTimeSeconds > relativeGranularity) {
            String[] lineSplit = line.split( regex: " ");
            double capturedData = Processing.DataProcess(lineSplit, dataTypes.get(dataTypeIndex));
            long finish2 = System.nanoTime();
            long elapsedTime2 = finish2 - start2;
            double elapsedTimeSeconds2 = (double) elapsedTime2 / 1000000000;
            //if elapsed time is greater than full granularity,
            // all data types have been checked and processed once and so the graph can be incremented
            if (elapsedTimeSeconds2 > granularity){
                graph.incrementTimeOne();
                graph.cp.repaint();
                graph.cp.revalidate();
                start2 = System.nanoTime();
            }
            //Check if current data type is "Rear EOG", process accordingly if so
            if (dataTypes.get(dataTypeIndex).equals("Rear EOG")) {
```

5.F RESTful API

For any online interactions a special class is used to interface with our Python Flask API and MongoDB servers. In this sample a request is sent from a button action listener to API to check the server and see if the provided account details match any of those contained in the server. If a match is found the API will respond with the “Authorised” string and the request to login will be filled. However, if the API times out or does not find a match the authorization will not be provided and the login request will fail.

```
public static boolean login(String username, String password, CallMaker callMaker) throws IOException {
    //create json body using given login info and Make a HTTP request to the the REST API login endpoint with genera
    String response = callMaker.MakeCall( endpoint: "login",  json: "{\n" +
            "\"username\" : \"" + username + "\",\n" +
            "\"password\" : \"" + password + "\"\n" +
            "}");
    return response.equals("Authorised");
}
```

6. Issues Resolved

6.A UI Design

Our initial design for the UI was a naive approach where we created frames and panels, filling them up with whatever contents were needed for the current panel. Then, when a new frame needed to be loaded, all the values that were currently being used would be passed to the new UI instance and the panel would be rebuilt. While disposing of the old UI and creating a new one wasn't an issue itself, passing values between classes constantly and adjusting the position of panel contents by pixels proved to create very hard to follow code very quickly. Examples of the previous UI implementation can be seen in the git commit history.

After a few weeks of working on the UI and learning how everything worked in Java (as it was our first time working with a Java UI) we decided to scrap everything and find a cleaner, more efficient way to build panels. After a couple of days of scouring the internet and our IDE we found that using the form layout from the JGoodies jar we could not only make our UI without needing to define everything for every object on the UI, but we could also use an interface to design that UI itself.

Taking what we had learned from our previous implementation and our new method of putting together UI pages, we managed to rebuild everything we had up to that point in a far cleaner, more legible way within a week. From there, we were actually able to follow what was going on in our own code and continue developing the application further in the weeks to come.

6.B Active UI Implementation

After having decided to use the form layout from the JGoodies jar, a new issue arose when adding anything to the interface that needed to be updated relative to results, or actively updated in real time. For example the Graph class which is partially displayed in the sample code above creates an instance of a XChartPanel which inherits, but is not considered a Java Swing base class. Because of this the XChartPanel can not be added to the interface in the form. Additionally the form does not allow for updates after it has been created. The combination of this means when loading a UI that necessitated a chart or any relative data we were unable to add a new panel to display these parts of the application.

What we found was that although panels could not be added or have contents added to them after they were created, the JScrollPane had a function that updated it's contents without adding a new panel to the form. This meant if we defined a JPanel, added all the contents we wanted to it, then called the JScrollPane.setViewportView() function and provided our contents panel we could actually update the interface as necessary.

6.C Muse Device Connection

While developing the functionality of the software that captured and processed the data from the Muse device, two main issues arose. The first issue became apparent when the GUI of the application seemed to crash or hang whenever data capture was being done in the first iteration of the build. It was discovered that the reason for this was because the process for running the API programs had to be run asynchronously to allow the UI code to continue to be run. The solution to this was encapsulating the processes for reading data into SwingWorker classes, which are designed to allow for multi-threading processes in a Java Swing application.

Another issue arose when this code needed to be unit tested. It was vital to ensure that code was running correctly during testing while also ensuring that no actual processes were being created, as this would be inappropriate and a poor testing technique. In order to circumvent this problem, code had to be refactored using dependency injection, where all of the untestable dependencies of the class are extracted into a helper class, which is then instantiated and initialised with the main class so it can be mocked. This ensures that the main class is running all of its code successfully without any probability of running dangerous or unstable code that shouldn’t be run in a unit testing environment.

7. Results

While our goals for everything we wanted to accomplish may not have been completed entirely, in the end we managed to produce at minimum a basic implementation of every feature we wanted the application to have. All the key features that are needed to make LiveFocus work and be recognizable as the application it was designed to be are there in one way or another. The interface could benefit from some improvements and some of the features only offer basic functionality, but given the time constraint of the project, that is to be expected. Given more time, a more appealing interface and more functionality would surely be added to produce a more complete version of the product.

From the project we both learned a good amount not only about the technologies used here, but also about the development process, time management, and allocation of resources. Both of us had our first interaction with Java UI development and most of the jars we utilized as dependencies here. As Cormac's Intra placement didn't occur, this was his first introduction to large scale testing and verification. In addition to this, both of us were repeatedly told by our supervisor not to spend too much time doing any one thing. "Work iteratively" is what we were told, meaning get a lot of small things working to some extent, then go back through them and improve them. That way we always had some work to show at the end. This really struck home with us when we tried for three days to fix one bug and realized how much time we had wasted that could have been spent on other functionality.

As there were a variety of different types of testing required for this the results of the tests can be found in the Testing Manual located in the docs/testing-manual directory.

8. Future Work

8.A Development

Following the submission for this project of what we have completed up to this point we would like to further develop the interactivity of the application without our current time restrictions. At the moment, while usable, the interface leaves a bit to be desired in terms of aesthetics, and that is something we would like to continue to tinker with until it looks nice and clean for everyone to use seamlessly.

Additionally there is some more functionality we would like to add to the system as well. Further development of the online interaction part of the system and more online user security features are some of the features on the list of things to continue working on. With this we would also like to expand the types of signals that a user can record and analyze as well as, if we can find someone with enough experience reading these signals, implementing some kind of system that analyzes the output and finds key points in a video for users with some kind of description of things that may have occurred at that point in time.

8.B Research

From our lack of knowledge about the values that are being recorded by the muse device and our application, we aren't able to determine exactly when a key action or loss of focus is occurring in a session. However, as we can demonstrate some correlation between predictable events and the results that appear on the graph, we assume that with more research and sample data we can be able to determine key events based on the result data alone. With the ability to determine key points based on data, we would be able to implement features that show users where they may want to edit their videos, without the user needing a high level of experience working with EOG and other muse signals.

With this assumption we would like someone who has more experience with EOG signals and other neural activity that the muse device records to help research and maybe point us in the right direction for extrapolating meaning from the signal recordings.

9. Sources

9.A Dependencies

1. Maven
2. XChart (3.8.0 jar): https://knowm.org/open-source/xchart/
3. JGoodies-Commons (1.8.1 jar): http://www.jgoodies.com/downloads/libraries/
4. JGoodies-Forms (1.8.0 jar): http://www.jgoodies.com/downloads/libraries/
5. JUnit (4.13.2 jar): https://search.maven.org/search?q=g:junit%20AND%20a:junit
6. Hamcrest (1.3 jar): https://search.maven.org/artifact/org.hamcrest/hamcrest-core/1.3/jar
7. Mockito (1.9.5 jar): https://mvnrepository.com/artifact/org.mockito/mockito-all/1.9.5

9.B References

1. Jiao, Yingying, et al. "Driver sleepiness detection from EEG and EOG signals using GAN and LSTM networks." *Neurocomputing* 408 (2020): 100-111.
2. Latifoğlu, Fatma, et al. "Detection of Reading Movement from EOG Signals." *2020 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*. IEEE, 2020.
3. Puthusserypady, Sadasivan, and Tharmalingam Ratnarajah. "Robust adaptive techniques for minimization of EOG artefacts from EEG signals." *Signal processing* 86.9 (2006): 2351-2363.
4. Wissel, Tobias, and Ramaswamy Palaniappan. "Considerations on strategies to improve EOG signal analysis." *International Journal of Artificial Life Research (IJALR)* 2.3 (2011): 6-21.
5. Zhou, Yajun, et al. "A hybrid asynchronous brain-computer interface combining SSVEP and EOG signals." *IEEE Transactions on Biomedical Engineering* 67.10 (2020): 2881-2892.