Control of Physical Objects Utilizing Brain Computer Interfaces

by

Nick Schmidt

A creative component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Lotfi Ben Othmane, Co-major Professor
David Jiles, Co-major Professor

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to thank all of those who contributed to my completion of this project. First, I would like to thank Dr. Lotfi ben Othmane for all of his guidance throughout my project as well as my graduate degree. His kind advice and technical knowledge kept me on the path to success. Additionally, I want to thank Dr. David Jiles for his time on my committee and his help with the project.

I would also like to thank my friends, in particular Brady Opsahl and Noah Halbur. Their constant input and humor kept me happy, motivated and on the right path. Without them my time at college would not have been nearly as fun or memorable.

Finally, I would like to thank my family. My brothers: Sam, Joey, and Paul were always available to talk and to make me laugh. My Mom, Colleen, and Dad, Mike, also gave me invaluable advice time and time again. Last, but not least, my dog Buddy's unconditional happiness always spread to me. Without my family's love and support none of this would have been possible.

ABSTRACT

Brain Computer Interfaces (BCI) are a rapidly growing and expanding field. BCIs are capable of creating an interface between the human brain and a computer. This paper lays out the required background knowledge to work with a BCI. It attempts to answer the question ”Can a BCI be used to control a set of real work objects?”. To do so, common methods of BCI were analyzed to create a reliable BCI. Then, an architecture was designed to allow for this BCI to be easily expanded to include functionality with as many devices as possible. The architecture developed as well as the logic and rationale behind it is laid out within this paper.

CHAPTER 1. Introduction

Brain Computer Interfaces are devices which are capable of interfacing between the brain and any computer device. The goal of this research was to create a general use Brain Computer Interface (BCI) that could operate with software as well as physical objects. To do this, an architecture was developed to allow for expansion into as many use cases as possible. Additionally, a proof of concept program was created to show that this architecture and BCI method worked.

1.1 Problem

The current problem with the field of BCI is the lack of a unified platform. There are many different commercially available devices to work with a BCI, but they are typically medically focused or only have first party software. This severely limits the capability for large scale development of BCI capable apps and devices. In order for BCI software to progress, a platform must exist which is capable of controlling physical and digital devices and which can be easily expanded.

To solve this problem, we set out to answer the question:

Can a user control a set of digital objects using their brainwaves?

An affirmative answer to this question impacts the way we interact with machines even if the techniques are limited in practice.

1.2 Approach

To address this research question, the first step taken was to begin researching available BCI devices. There are many commercial products available, including the Emotiv Epoc+ used for the project. Next, the problems with each of these BCI were researched to find what needed to be solved. A machine learning and data collection method was created to attempt to match performance with the available devices own prediction methods. Finally, this methodology was tested to see if it could accurately control real world objects and a general use architecture was developed.

1.3 Contribution

The major contributions of this thesis are :

- Developed an algorithm that allows a user to control a digital object using their brainwaves.
- Developed a framework that allows for the collection of Electroencephalogram (EEG) and other BCI data from devices and allow for the easy development of applications that use this data to control digital objects.

1.4 Organization

This paper is divided into 8 chapters. This chapter describes the paper itself. Chapter 2 describes necessary background information on BCIs and data collection from the brain. Chapter 3 describes in depth the steps taken during the project to answer the research question. Chapter 4 describes the different types of plugin architecture. Chapter 5 lays out the architecture created to solve the research question. Chapter 6 lays out the system created to prove the architecture works and the research question was answered. Chapter 7 lays out the evaluation of the machine learning methods and architecture.

CHAPTER 2. Background

The following section will highlight the relevant background information in the research of brain computer interfaces. For the implementation of a proof of concept of this architecture, an EEG based headset was used. This section will also cover the methods used to create this interface, highlighting the processes used for creating usable data.

2.1 Methods of Brain Data Extraction

There are several methods for reading information from the brain: non-invasive, partially invasive, and invasive [4].

A non-invasive method, which is the most common, involves reading information from the brain from outside of the skull. This usually involves the placement of scanners on the head of the subject. The most common implementation of a non-invasive brain computer interface is the electroencephalogram [5]. The main benefit of non-invasive methods is that the process is safe and easy. All that is required is a scanner and there is no advanced procedure required to use the sensors. The second major benefit is that these types of scanners are cheaper than other methods. The size and quality requirements for a scanner placed outside of the skull are also significantly less than internal scanners [5]. Larger scanners of lower material quality can be used when the scanner is external.

The main issue with this method is signal positioning. Since all data is measured from sensors outside of the skull, locating where these readings take place is difficult. There are a few mitigation methods, but they are usually computationally expensive [6]. The second issue with this method of data recording is interference. Since this data is read from outside of the skull, there is interference from muscles throughout the head as well as the insulation and interference of the skull. This can cause readings to be inaccurate and can limit the ability to read signals of neurons deeper within the brain [7].

The second method for reading information from the brain is the partially invasive method. This method is mainly focused on removing any interference from muscles and the skull. It involves placing scanners within the skull and on top of brain matter but does not involve inserting anything within the gray matter [8]. Sensors can be placed on the relevant portions of the brain to focus data readings. Therefore, it is easier to identify the portions of the brain that are producing a signal without interference from muscles or the skull. This helps to increase the accuracy of the data being read.

The final method for reading information from the brain is the invasive method. This method requires sensors be placed within the grey matter of the brain. This is by far the most accurate method of data collection, but also highly intrusive [6]. This method allows for highly accurate readings from all sections of the brain, including those deep within the brain. This allows for much more accurate 3D modeling of signal locations within the brain. Additionally, sensors can be highly focused within known areas of the brain to focus on things like vision or hearing. The final major benefit of invasive methods of data collection is the minimal level of noise from outside sources [9].

2.2 EEG Based Brain Computer Interfaces

Currently, the most common method for creating a non-intrusive BCI (Brain Computer Interface) is through using an EEG. An EEG, short for Electroencephalogram, reads in raw voltages from the brain. It is non-invasive and is also the most commercially available. Traditionally, the EEG was used within medical settings to diagnose issues within patients. However, cheaper methods have allowed for commercial use of these products.

The process for using an EEG follows. Sensitive electrodes are placed on the head of the individual. These electrodes measure minute voltages generated within the brain in micro volts. EEGs can utilize any number of electrodes with a higher number of electrodes allowing for a more accurate signal placement. Due to interference from the skull and the 3D nature of the brain, pinpointing the exact position of a signal is difficult. Probabilistic methods can estimate the locations of signals based on the activation of surrounding electrodes. Due to its simplicity, the EEG has several benefits over other methods of data collection [10].

The first benefit of the EEG is that it is easy to use. The placement of sensors on the head requires minimal setup and no advanced training. The second major benefit of an EEG is the low cost afforded by the simpler parts. Many other methods of data collection require advanced and expensive machinery, such as the MRI. The third advantage of the EEG is the temporal accuracy of the readings. Since the electrodes read the voltage present at their location, near-instant recording of this data is available. There is no requirement of advanced processing or measurement to receive this raw data. Unfortunately, these benefits are not without their consequences[7].

The major issues with EEG based voltage recordings are mostly related to accuracy. The first issue is related to muscle interference. Electrodes are placed on top of the head and read in raw voltages, any muscle flexing on the subject's head results in a reading on the electrode. This can cause inaccuracies and inconsistencies throughout the data. The second major issue is the lack of location-based data. Complex probabilistic methods are required to attempt to estimate locations. Additionally, voltages read at one electrode may not necessarily come from that section of the brain. With many different parts of the brain generating signals, these signals will cause changes on all electrodes throughout the head.

Overall, the EEG is the most accessible data collection method. The issues related to EEG-based data collection aren't necessarily a negative in all use cases. For instance, the limitations in knowing the location of the signal does not affect differentiating signal groupings. If the patterns are unique, the details of how these patterns are unique aren't as important.

2.3 Brain Waves

The previous section described EEG voltage collection. This data has many different uses but can vary greatly over time. To create more consistent readings, this data can be transferred into brainwaves. By collecting these signals over a chosen time period, the raw voltages can be converted into a measure of wave-forms. These waves are referred to as brain waves. The conversion of this data is performed using several available methods, these will be discussed in Chapter 3. Much research has been performed on brain waves and their corresponding thought patterns [11].

The types of brainwaves were split into a few different categories. These categories are Delta, Theta, Alpha, Beta, and Gamma. The first type of brain wave, delta waves, consist of those waves which are of the frequency .5 – 4 Hz. These very low frequency brain waves are often associated with meditation or dreamless sleep. Theta waves are those waves which range from 5 – 8 Hz and are often associated with drowsiness or dreaming. Alpha brain waves are those waves which range from 9-14 Hz and are often associated with a relaxed or resting brain. Beta brain waves are those brainwaves that range from 15-30 Hz and are often associated with an alert or working brain. Finally, gamma waves are all other waves and are associated with a highly active brain [12].

The concept of a power band is used to measure the prevalence of different brainwaves. This concept essentially measures the prevalence of each type of wave within the brain. This operation utilizes the spectral density operation. This operation calculates the amount of power each frequency range is expected to contribute to the overall range of frequencies.

2.4 Emotiv Epoc +

The Emotiv Epoc+ was used for the implementation and testing of our theory. This headset is a commercially produced EEG device. The device has 14 electrodes which provide EEG data wirelessly. The internal EEG readings take place at a rate of 2048 readings per second and are provided to the user at a rate of either 128 or 256 samples a second. The device also has a built in gyroscope and accelerometer.

The Emotiv Epoc+ electrodes are placed at the following zones within the head: F7, F3, AF3, AF4, F4, F8, T7, T8, P7, P3, P4, P8, O1, and O2. Nodes P3 and P4 are only for reference readings and will not provide EEG data to a user. These sensor locations are provided by Emotiv in the brain map in Figure 2.1.

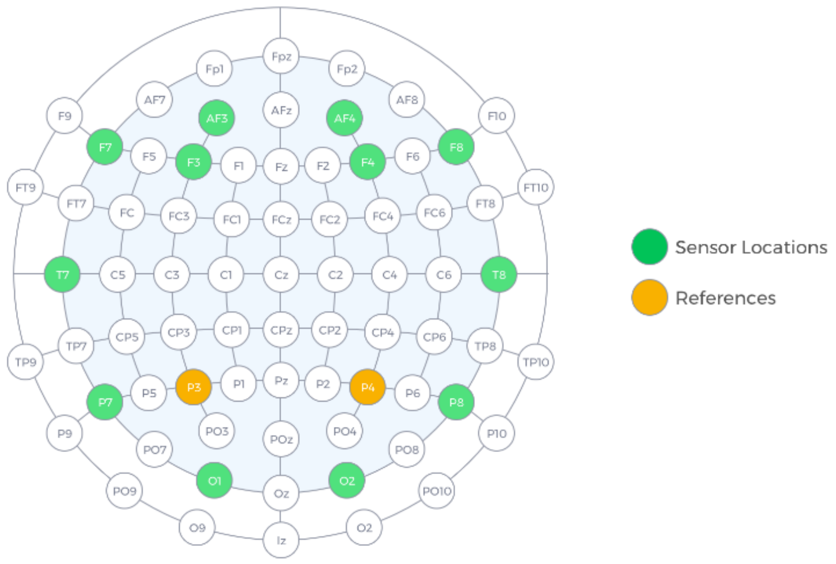
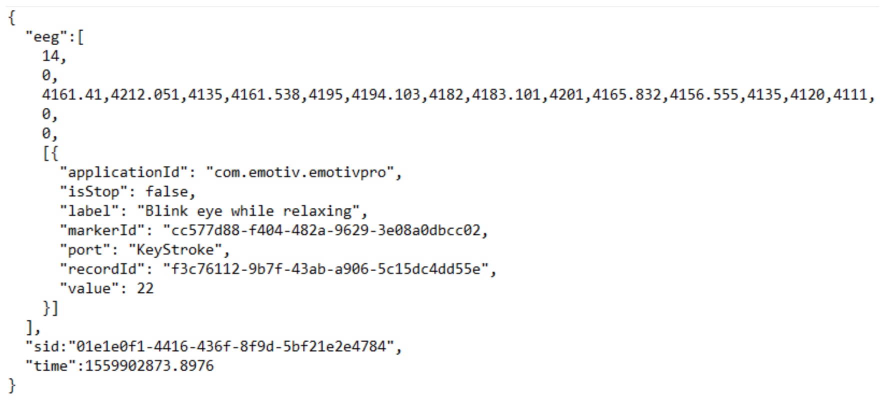


Figure 2.1 Emotiv Epoc+ Sensor Placement Map[1]

```
[
        "COUNTER",
        "INTERPOLATED",
        "AF3","F7","F3","FC5","T7","P7","O1","O2","P8","T8","FC6","F4","F8","AF4",
        "RAW_CQ",
        "MARKER_HARDWARE",
        "MARKERS"
]
```

Figure 2.2 Emotiv Header Data[1]

The Emotiv Epoc+ utilizes an application to pass any data to a user. The user must have an account with their service in order to send and receive data. Once an account is created and the application is downloaded, a user can send and receive information from the headset by first submitting a JSON request to the localhost port provided by the application. The application will then handle all back-end communication. After this request is received by the localhost port a header describing the data's format will be sent followed by the data on a preset interval. The format of the request utilized for subscribing to EEG data is below. Figure 2.2 shows a sample of the header data sent as the initial message from the software. Figure 2.3 shows a sample of a typical EEG body message received from the software on a regular interval. Figure 2.1 shows the Emotiv Epoc+ Sensor Placement Map.[1]



```
{
  "eeg":[
    14,
    0,
    4161.41,4212.051,4135,4161.538,4195,4194.103,4182,4183.101,4201,4165.832,4156.555,4135,4120,4111,
    0,
    0,
    [{
      "applicationId": "com.emotiv.emotivpro",
      "isStop": false,
      "label": "Blink eye while relaxing",
      "markerId": "cc577d88-f404-482a-9629-3e08a0dbcc02,
      "port": "KeyStroke",
      "recordId": "f3c76112-9b7f-43ab-a906-5c15dc4dd55e",
      "value": 22
    }]
  ],
  "sid:"01e1e0f1-4416-436f-8f9d-5bf21e2e4784",
  "time":1559902873.8976
}
```

Figure 2.3 Emotiv EEG Data[1]

2.5 Current BCI Deficiencies

Currently, few commercial offerings are available in the BCI space. Those that are available are typically geared towards medical use. However, a few commercial products with a focus on a more general audience were identified and some of the issues with these platforms were targeted for improvement in this paper. One of these devices was the previously mentioned Emotiv Epoc+. A few other products identified were the NeuroSky, OpenBCI, and OpenVIBE. Each of these products will be discussed below.

The first two types of devices available are the Emotiv Epoc+ and NeuroSky devices. These headsets both offer a similar style of service. The devices come with a software pack or interface that allows for a user to communicate with the device for their own software's uses. Emotiv elects to use a localhost webserver to allow for the extraction of data, while NeuroSky includes a library for the development of apps which will appear on the NeuroSky app store. While both of these offerings work from a development standpoint, they suffer from the requirement that a proprietary device is used. There is no way to develop for a general EEG and any piece of software would have to develop specifically for that brand of EEG device.

The second style of device available is OpenBCI. OpenBCI is targeted at allowing for open use of EEG software. The company sells separate Arduino boards for use with electrodes. A series of python libraries are then offered by the company to allow for interfacing with these devices. This allows for a more open environment than the offerings from Emotiv or NeuroSky, but still runs into compatibility issues with devices which do not implement the OpenBCI libraries.

The final solution available is OpenVIBE. OpenVIBE seeks to solve the major issue brought up with the previous styles of device, the lack of compatibility. OpenVIBE does so by implementing a GUI coding interface with pre-made blocks which handle various operations. Users can implement their own "boxes" using either python or C++. This allows for developers to create their own logical flow for a BCI device and tie it into their own software. However, one of the major pitfalls of OpenVIBE is its use of a graphical coding interface and a relatively complex method of coding support without the use of the GUI. The use of a graphical coding interface requires that developers learn to use this GUI and brings developers outside of their own environment. This can be an issue for some development projects where developers would prefer that the solution be included in their own software. Additionally, a developer must understand the flow of a BCI device in order to develop the core implementation of the software utilizing the GUI.

Overall the issues with these different styles of implementation are similar. The first two styles have issues with implementation of devices outside of their scope. OpenBCI devices cannot easily work with NeuroSky or Emotiv devices and NeuroSky and Emotiv devices require coding specifically for their devices. The other issue identified is with OpenVIBE which can require complex understanding of the flow of a BCI to implement. Additionally, to easily work with OpenVIBE a developer must move outside of their environment and work within the OpenVIBE GUI code editor. The solution proposed by this paper will attempt to reduce these issues.

CHAPTER 3. Solution Approach

This chapter describes the approach used to test whether a single object could be controlled using their brainwaves. The first step in the approach involved researching details for common methods of EEG interpretation. The second step proved that these readings could be categorized using machine learning. The third step determined whether these categorizations would be fast and efficient enough to allow for realistic and real time control over a real world object. The final step was the expansion of these categorization techniques into an extendable architecture to allow for multiple input devices, methods of communication, and output devices.

3.1 EEG Interpretation Method

There are several common methods for extracting usable data from EEG voltage signals. Three of the most effective and common methods of EEG transformation are the fast Fourier transform, auto-regressive methods, and wavelet transform methods. Each of these classification methods has their own advantages and disadvantages.

The first and most common method of EEG interpretation for BCI applications is the FFT. The main advantage of the FFT is that it is computationally lighter than other methods. However, this method is more susceptible to noise and has issues interpreting quick changes in data. A signal must be held for a consistent period of time for the FFT to identify it. The FFT also suffers from issues with quick spikes in a signal. This can happen with an EEG signal due to muscle or eye movements triggering electrical signals in the muscle. These disadvantages are counterbalanced with the fact that the FFT requires no heavy computation or modeling. The FFT does not require a special model for each input signal[13].

Table 3.1: Advantages and Disadvantages of EEG Analysis Methods.

Model	Advantages	Disadvantages	Analysis	Suitability
Fast Fourier Transform	1. Good tool for stationary signal processing 2. It is more appropriate for narrow-band signals, such as sine wave 3. It has an enhanced speed over virtually all other available methods in real-time applications	1. Weakness in analyzing non-stationary signals such as EEG 2. Doesn't have good spectral estimation and cannot be employed for analysis of short EEG signals 3. FFT cannot reveal the localized spikes and complexes that are typical among epileptic seizures in EEG signals 4. FFT suffers from large noise sensitivity, and it does not have shorter duration data record	Frequency Domain	Narrow-band, stationary signals

Continued on next page

Model	Advantages	Disadvantages	Analysis	Suitability
Wavelet Transform	1. It has a varying window size, being broad at low frequencies and narrow at high frequencies 2. It is better suited for analysis of sudden and transient signal changes 3. Better poised to analyze irregular data patterns that is, impulses existing at different time instances	Needs selecting proper mother wavelet	Both time and freq. domain and linear	Transient and stationary signal

Continued on next page

Model	Advantages	Disadvantages	Analysis	Suitability
Auto-regressive	1. AR limits the loss of spectral problems and yields improved frequency resolution 2. Gives good frequency resolution 3. Spectral analysis based on AR model is particularly advantageous when short data segments are analyzed, since the frequency resolution of an analytically derived AR spectrum is infinite and does not depend on the length of analyzed data	1. The model order in AR spectral estimation is difficult to select 2. AR method will give poor spectral estimation once the estimated model is not appropriate, and model's orders are incorrectly selected 3. It is readily susceptible to heavy biases and even large variability	Frequency Domain	Signal with sharp spectral features

The second common method of EEG analysis is the auto-regressive method. This method deals with spikes in signal much more readily than the FFT. Additionally, the frequency resolution on an auto-regressive method is much higher. This allows for more accurate readings on the frequency.

The major issue with an auto-regressive model is the fact that it requires a predetermined model which can become inaccurate. Additionally, the AR method is more computationally intensive than the FFT[13].

The third common method of EEG analysis is the wavelet transform. This method deals with EEG data accurately, but requires the selection of a clear mother wavelet to analyze the data. It is also more computationally intensive than the FFT method.

In selecting which method to utilize for the categorization of commands, considerations were made based on the common issues related to each method. The first major concern was speed. Developing software that wouldn't allow for use across many types of hardware would be a hindrance to the research and future work. There was also a consideration placed on ease of use. Requiring modeling and complex data collection for each individual user would not be feasible and would also go against the spirit of the research. The idea is to allow for the accessible and easy utilization of brain-waves for control of objects.

The final criteria was the requirement for accurate data. In a medical setting, the frequency resolution is highly important and must allow for the observation of minute changes and quick jumps in frequency. However, for the development of a BCI, this does not need to be as heavily considered. The main focus of the BCI developed in this paper is to allow for the control over several real world object using a set of trained commands. This does not require as high of a frequency resolution, as the only concern is matching patterns and not analyzing them.

Due to these considerations, the FFT was selected. The data collection rate of the Emotiv Epoc+ is 128 signals per second, computing an FFT on this data is already CPU intensive. These signals must also be passed off to other programs at later steps of development. Computationally intensive methods such as AR and wavelet transforms should be avoided.

After the initial interpretation method was selected, this had to be built out to allow for the machine learning method to work with the data. One way to make FFT data more readable and consistent is through a conversion to power band data. In order to perform an FFT, enough samples must be collected for the algorithm to work. In our case, 256 samples were utilized. This method also requires noise be removed from the data. This is can be accomplished with the hanning window and detrend methods, which will emphasize the major points of the wave and remove smaller ripples near the outsides of the waveform. Finally this data is squared to retrieve the power band data.The final stage is to iterate over the frequencies and to add the waves to the correct bucket. This algorithm is described below. This method is provided by Emotiv and is also heavily utilized in the BCI community for the fast and easy interpretation of data[13].

Algorithm 1: Power Band Conversion

```
Result: Power Band Data By Bucket
EEGData = new Array;
while Receiving Data do
    EEGData.insert(Data);
    if length(EEGData) = 256 then
        Detrend(EEGData);
        Hanning(EEGData);
        FFT(EEGData);
        PowerDensity = FFT * FFT;
        for Frequencies in PowerDensity do
            Add frequency to corresponding power band bucket;
        end
        Print bucket information to a log file;
        EEGData.remove(Oldest 16);
    end
end
```

3.2 Categorization

The second major step in the research approach was selecting and implementing a categorization method for the data. These approaches and steps are described below.

3.2.1 R Based Programming

The initial steps in the process involved testing several common and simple machine learning methods on large files of converted EEG data. This data was converted using both the pre-built

Emotiv method as well as the power-band method described in section 3.1. However, according to available Emotiv documentation, these methodologies are similar if not the same.

To begin, the data was brought into a simple R program. The data was numbered based upon the command it corresponded too. Ten samples were pre-recorded with five neutral samples and five samples issuing a command. The neutral commands were numbered zero and the command was numbered one. A separate set of ten commands was recorded to utilize for testing the methods. The first few methods applied to the data were to determine if a simple solution would work. Initially, a regression was utilized in R. It became very clear that this method was not working for the sheer number of inputs. In total, there were 70 data nodes which had to be utilized for classification. A clustering approach suffered similar issues, failing to categorize even 50 percent of the models correctly.

3.2.2 Keras Programming with Python

Due to the issues encountered almost immediately and the major lack of accuracy using these methods, it was determined that a library capable of using complex neural nets and deep learning would be utilized. A neural network was selected for this task due to its capability of finding non-linear relationships between inputs and outputs. The major downside to a neural network is the requirement for a large dataset [14]. However, the volume of data that could be easily recorded through the headset mitigated this issue. The most readily available and powerful of these libraries was Keras, available in python. Keras utilized tensor flow to leverage a graphics card's computation power for the categorization of commands. This new process is described below.

Keras is a powerful and open source machine learning solution [15]. It utilizes tensor flow, which leverages the graphics card in the computation of machine learning data. This is powerful for large and complex data sets and allows for the model to configure itself to most accurately categorize data. This section will describe the deep learning methodology and describes the method arrived at based on experimentation.

Deep learning is a complex method of machine learning. Deep learning is modeled on the decision making observed within an animals brain. A deep learning network consists of several nodes, referred to as neurons. Based upon previous inputs, these neurons will either activate or deactivate. The first layer in the network is an input layer with one node to correspond to each input signal. In the case of the data we are using, there will be 70 input nodes. Five brain wave buckets in the power-band data for each of the 14 data recording electrodes. Next, this data is passed into hidden layers within the network. These hidden layers each hold a predefined number of nodes or neurons. During training, these hidden layers continually reconnect with different hidden layers' nodes and with nodes within their own layers. Each hidden node continually adjusts its parameters to formulate the requirements for when the node should activate. This is determined based upon the connections with previous nodes as well as a weight which is assigned to these connections to determine how important they are in the final determination. This process is repeated many times over the pre-classified test data until the model goes through the preset amount of iterations, in our case 10 [16].

The final model arrived at for the use in the machine learning method was relatively simple with one input layer, 2 hidden layers, and 1 output layer. This model can be easily adjusted to allow for tweaks based on the accuracy and needs of other users, However this model proved accurate on our test models. A full evaluation on the tested models is provided in Chapter 7.

3.3 Real World Control

The third stage in the approach was to test the machine learning methodology in real time. In order to provide a proof of concept, a simple program was developed to attempt to control a lamp utilizing EEG. This lamp was connected to a Raspberry Pi running a local webserver. This webserver would listen on the localhost for a command sent from the computer running the EEG software and would toggle a relay to enable electrical signal to flow to the lamp. In order to do so, the following process was followed:

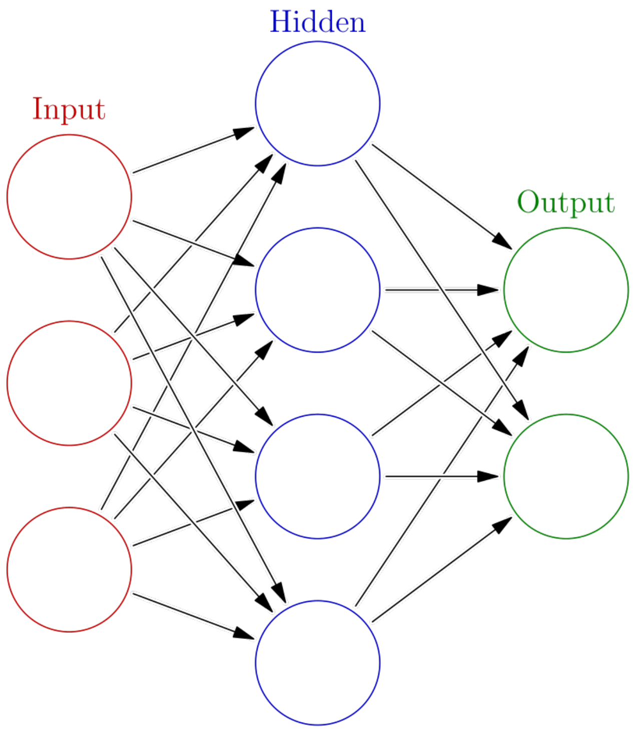


Figure 3.1 EEG Interpretation Method Analysis[2]

1. Record five ten second long logs containing the raw EEG data for sending no command (neutral)

2. Record five ten second long logs containing the raw EEG data for sending a lamp toggle command (toggle)

3. Convert these logs into power band data using the FFT algorithm described above

4. Train a model using the described method in Keras

5. Import this model into a new program that will listen to EEG data and utilize the model to determine when a command will be sent

In order to accomplish the final step, a new algorithm had to be developed to work with the live data. This algorithm works very similarly to the power band data but adds additional steps for working with the live data and for the categorization step.

Algorithm 2: Command Interpretation

```
Result: Command
import keras model;
while Receiving Data do
    EEGData.insert(Data);
    if length(EEGData) = 256 then
        Detrend(EEGData);
        Hanning(EEGData);
        FFT(EEGData);
        PowerDensity = FFT * FFT;
        for Frequencies in PowerDensity do
            Add frequency to corresponding power band bucket;
        end
        keras.predict(PowerDensity);
        EEGData.remove(Oldest 16);
    end
end
```

This categorization algorithm proved to work, as is shown in Chapter 7. Additionally, I was able to have a usable level of accuracy for playing Pacman. This test was created using MAME, an open source arcade machine emulator. Additionally, Pyvjoy and Vjoy were utilized. Vjoy is an open source software which allows for control of a virtual joystick on windows machines. Pyvjoy is a python library which allows for interaction with Vjoy using only python. A model trained for four inputs was also required, one for each joystick direction. This Pacman classification could be run at the same time as the lighting classification. This proved that brain data could be used to control objects in the real world.

3.4 Requirements

A major requirement for a solution was that it must be modifiable and expandable to allow for many different use cases. This would theoretically allow for any object in software or in the real world to be controllable using this headset. In order to do so, we investigated several common architectural solutions to the problems of modifiability and usability. Several solutions were proposed, including things likes micro-services or server-client information, which is similar to the

Emotiv approach. However, the approach which stood out the most was the plugin architecture. This architecture will be discussed in detail in the next chapter.

CHAPTER 4. Plugin Architecture

In the course of the research into modifiable and easily usable architectures, it was determined that a plugin architecture would best fit the needs of this project. This selection was made due to the easy addition of new functionalities from third party developers when compared to other architectures. This worked well to mitigate the issues encountered by other BCI implementations. This section will describe the benefits of a plugin architecture. It will also describe some of the examples of how this architecture is implemented. The next section will focus on the implementation developed for the use as a BCI plugin architecture.

4.1 Architecture Description

A plugin architecture is an architecture whose core functionality can be expanded or modified based on the insertion of new classes or objects. These classes or objects must fit a predefined definition for the structure of a plugin. These are dependent upon the software being expanded, but usually involve some type of scripting language or .XML file. There are many different ways in which a plugin-based architecture can be implemented. We will begin by describing two very popular implementations of a plugin architecture, the WordPress architecture and the Eclipse architecture. Figure 4.1 demonstrates a common plugin implementation. A core piece of code is expanded through the use of plugins, developed by the creator or other users. These plugins can be added and removed without affecting the core itself [17].

4.1.1 WordPress Architecture

WordPress is a popular website development and hosting service. A user of the service can create their own website using the tools available. However, this website also includes the possibility of expanded functionality through the use of plugins. These plugins can modify or add to the behavior

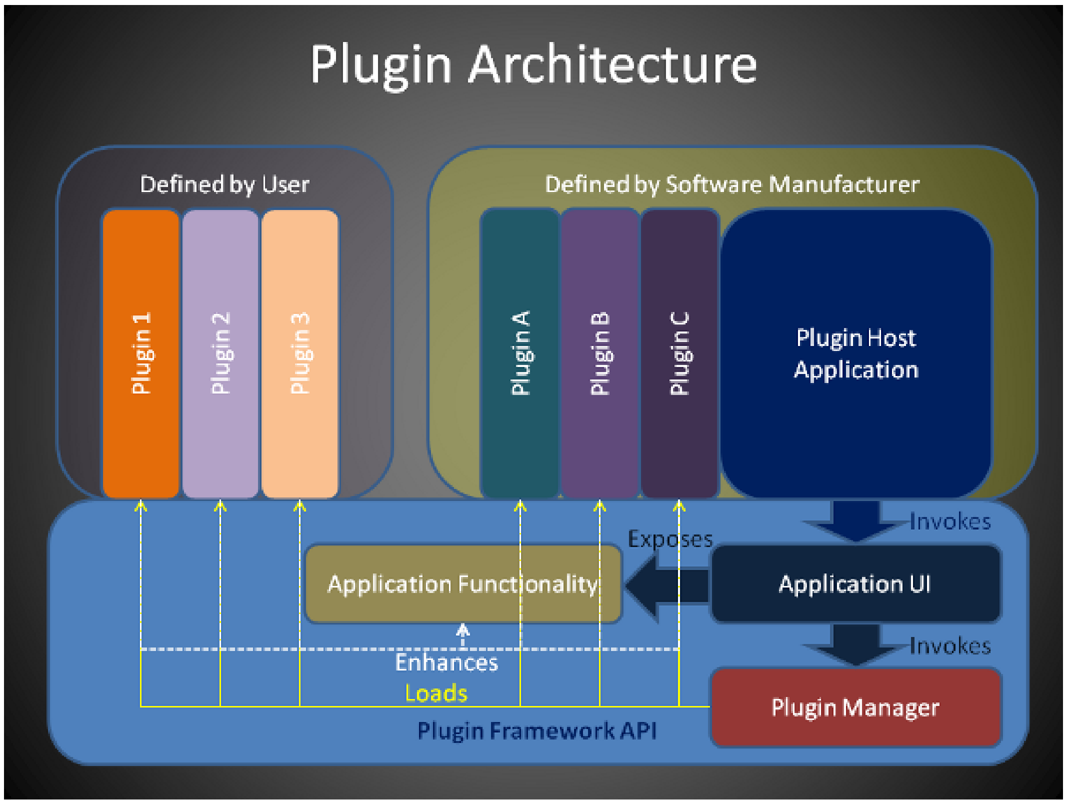


Figure 4.1 Plugin Architecture Example[3]

of a website. Some popular plugins include plugins to seamlessly upload podcasts to your site as well as plugins to modify your site to be a web store.

These plugins work by utilizing the idea of a "hook" in the WordPress site. These hooks are predefined areas which allow for user defined code to run. A plugin developer writes code to access a piece of software when a particular hook happens. For example, a hook defined by WordPress is "the_post". Registering your plugin to wait for this hook will allow your plugin to interact with a post object before it is sent to the site. There are other actions that allow for more simple actions, such as "wp_loaded" which simply runs your code after the website is loaded.

This method of plugin interaction is simple to understand and relatively powerful. Through the use of hooks, WordPress allows developers to make modifications to many portions of the website. However, this method is less effective when applied outside of the website context. There are many cases in which you would want your plugin to allow for complex data interaction and usage of user defined objects in a more robust way. This is one of the major downsides of utilizing the WordPress style of plugin outside of website formats[18].

4.1.2 Eclipse Architecture

The Eclipse architecture builds on the idea of a plugin architecture, and expands it to the entire program. As opposed to WordPress, a majority of the Eclipse core functionality is implemented as plugins. The design of the core program is very minimal, with plugins expanding out a nearly non-existent core functionality. This architecture is more complex to fully implement, but allows for robust control over plugins and nearly limitless expansion.

In order to implement this idea, the Eclipse architecture utilizes a few key concepts. First, instead of utilizing "hooks" as in WordPress, an Eclipse architecture utilizes extension points. These extension points are defined within each individual plugin through the use of an .XML file. This file lays out the required fields which an extending plugin must implement, as well as the format of any data that will be transferred between the two. This is typically implemented as a Java class which can then be used to call specific needed methods. The plugins themselves are then defined within .XML files which describe some basic information about the plugin. The implementation of these plugins is then completed using a Java class [19].

The advantage of this is clear. A user can easily add any plugins they desire onto the provided extension points and allow for complex interaction between plugins. Additionally, the core Eclipse platform is implemented as plugins, allowing for theoretical modification of the core Eclipse functionality. This allows for a high level of customization and modification of the available functionalities for the software.

4.1.3 Combining the Ideas

Each of these different architectures has a few strengths. The first major strength of the WordPress architecture is that it allows for the simple implementation of plugins based on a website. The architecture doesn't need complex interaction because the data being transferred is all related to the website itself. This is different with an Eclipse plugin where a user may have many complex objects which are being created and passed between plugins. The benefit of the WordPress implementation is its simplicity and ordered execution. The WordPress core controls the flow of the logic through it's activation of various hooks, as opposed to the Eclipse architecture which hands off a certain level of control to the plugins which interact with each other.

This method of communication has some clear advantages in the world of BCI. A core program could manage input from the interface device and activate available plugins through the use of hooks. This would ensure that data is managed by the core program and that plugins are responding to data and not holding up the execution of the remaining program. However, the major downfall of the hook method is the lack of complex interaction of user defined objects. This is necessary for the use of a BCI architecture which could allow interaction with many different plugins meant to do unknown tasks. The design of such a software should allow for more complex Eclipse style plugins to be meshed with simple WordPress style hooks. This is the goal of the architecture that was developed and is defined in the next section.

CHAPTER 5. Architecture Implementation

This chapter outlines the architecture developed to address the concerns of a BCI environment. This architecture is designed utilizing the ideas presented by WordPress and Eclipse and with the issues encountered by other solutions in mind. To start, this section will describe the concepts of this architecture. It will then break down each of the elements of the architecture as well as how new plugins can be implemented. Specific implementation details can be found in Appendix A. This architecture utilizes three types of plugin: interfaces, devices, and plugins.

5.1 Layout

The core of this architecture is defined based upon the Eclipse architecture. However, due to the nature of a BCI, there must be plugins which drive the rest of the application. To facilitate this need, the WordPress architecture was leveraged. Two new types of plugins were developed, the device plugin and the interface plugin. A device plugin will be responsible for communicating with a BCI, in this case the Emotiv EEG headset. The definition for devices will be defined in the Device section. These devices have a generic interface, similar to those available in WordPress. Interface plugins will connect to these generic hooks within the device plugins and wait for input. Generic plugins will then act based upon any interfaces which they extend.

5.2 Plugin

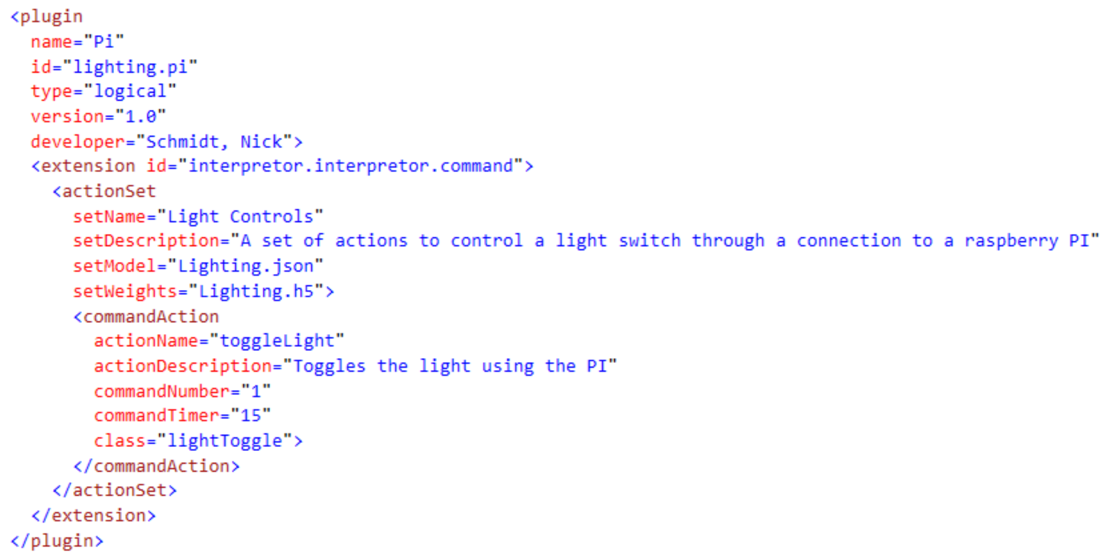
This section outlines the implementation of the generic plugin in this architecture. The generic plugin will make up a majority of plugins designed within this architecture. Generic plugins are capable of defining their own extension points and are also capable of implementing extensions to as many plugins as desired. A plugin consists of three types of files, plugin definition files (.xml), implementation files (.py) and extension definition files (.ext).

A plugin file should include any required functionality for the core of a plugin. Logic related to specific extension points should be placed into extension point implementation files, which will be touched on later. A plugin implementation may contain no functionality so that these functionalities can be passed off to extension points. These extension points as well as any extensions implemented by the plugin should be defined within the plugin definition file.

A plugin definition file defines the details of a plugin. An example of this file is included in figure 5.1. The file begins with enclosing plugin tags to allow for parsing by the core program. The plugin then identifies some basic facts about itself including its name, id, what type of plugin it is, the plugin version, and the developer.

The next portion of the plugin definition file is the extension list. Each extension point extended by this plugin must be defined within the extension tags. To begin an extension, the id of the extension point must be entered. Next, any additional elements required by the extension point must be enclosed within their own tags. In the example from figure 5.1, we can see that there are two elements required by the extension point interpreter.interpreter.command. The extension point concept will be expanded upon in the next section. What is required to understand the plugin definition file is the knowledge that within each tag we pass an extension point's elements the required variables. For example, we are passing the actionSet element the variables setName, setDescription, setModel, and setWeights. We are also passing the actionSet element a different element, known as commandAction. This commandAction element is being given the variables actionName, actionDescription, commandNumber, commandTimer, and a class which implements the required interface.

The final requirement within a generic plugin definition file is the inclusion of any extension points. The plugin provided within figure 5.1 does not have any extension points built into its functionality, so an example extension point was created. To declare an extension point, the extension-point tag is used. This definition must include an ID for the extension point, as well as a name and a schema. This schema will point to the file which holds the definition for that extension point.



```
<plugin
  name="Pi"
  id="lighting.pi"
  type="logical"
  version="1.0"
  developer="Schmidt, Nick">
  <extension id="interpretor.interpretor.command">
    <actionSet
      setName="Light Controls"
      setDescription="A set of actions to control a light switch through a connection to a raspberry PI"
      setModel="Lighting.json"
      setWeights="Lighting.h5">
      <commandAction
        actionName="toggleLight"
        actionDescription="Toggles the light using the PI"
        commandNumber="1"
        commandTimer="15"
        class="lightToggle">
      </commandAction>
    </actionSet>
  </extension>
</plugin>
```

Figure 5.1 Generic Plugin Definition

5.3 Extension Points

The developed architecture also implements a form of Eclipse style extension points. These extension points are defined by each plugin within an extension point schema file (.ext). An extension point schema file for the extension point referenced by figure 5.1 is available in figure 5.2. Similar to the plugin definition file, this extension point file begins with a schema tag to allow for the core program to parse it. The next tag is an element tag. This defines the elements that can be implemented by any plugin wishing to extend this point. The first element in any extension point is the extension tag. This tag lists which elements must be implemented by default. In this example, any extension wishing to extend out the extension point must include at least one actionSet element.

Next, a second element is listed. This is the required actionSet element. This actionSet requires the use of a commandAction element and also requires several arguments. The argument tag defines these arguments name and type, as well as whether they are required. A similar definition is created for the commandAction element, but as you can see from the example figure, there are no required elements for a commandAction.

The next part of an extension point is the extension point implementation file. For many plugins, this is where a majority of the functionality will go. This file must have the same name as the extension definition file and will contain the python code for this extension point. This code will execute based on the methodology implemented by the developer. It is possible for an extending plugin to cause the extension point to take action but it is also possible for an extension point to cause its extensions to take action. Each extension point must also implement an extensionHandler() method. This method will be called as the generic communication method between an extension point and it's extension, performing the implemented behavior for each extension.

The final part of an extension point are the element implementation files. These files can range in usability. Some definition files may contain nothing but the expected methods that an element must implement and pass the functionality to the extension point. Other elements may be fully implemented and function based solely on the attributes passed into the extension point from an extending plugin. These decisions are up to the plugin developer.

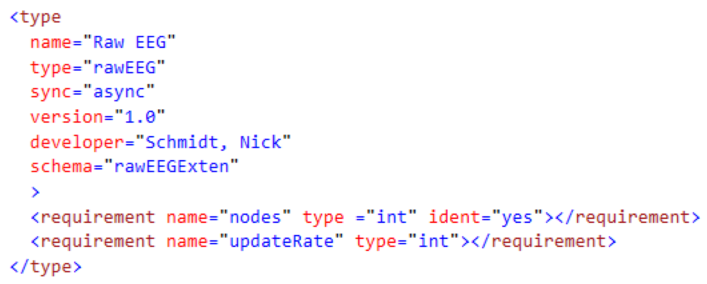
```
<schema>
  <element name="extension">
    <required>
      <elementReq ref="actionSet" min="1" max="0"/>
    </required>
  </element>
  <element name="actionSet" implementation ="actionSet">
    <required>
      <elementReq ref="commandAction" min="1" max="0"/>
    </required>
    <argument name="setName" type="str" req="yes"></argument>
    <argument name="setDescription" type="str" req="no"></argument>
    <argument name="setModel" type="str" req="yes"></argument>
    <argument name="setWeights" type="str" req="yes"></argument>
    <argument name="class" type="string" req="no"></argument>
  </element>
   <element name="commandAction" implementation ="commandAction">
     <argument name="actionName" type="str" req="yes"> </argument>
     <argument name="actionDescription" type="str" req="no"> </argument>
     <argument name="commandNumber" type="str" req="yes"></argument>
     <argument name="commandTimer" type="int" req="no"></argument>
     <argument name="class" type="string" req="no"> </argument>
   </element>
</schema>
```

Figure 5.2 Extension Point Definition

5.4 Device Plugins

The next type of plugin available is the device plugin. This type of plugin was created specifically for working with a BCI. The plugin definition is divided into two parts. The first is the creation of a type. These types will hold the extension point definition for all devices which utilize this type. This will force consistency across all implementations of this type. For example, a new EEG headset could be used with the software and no changes would have to be made to other plugins. An example type definition is supplied in figure 5.3. This definition must include the name, the type, the version, the developer, and a schema describing the extension point definition. Additionally, the type can include required arguments for an interface wishing to access the device. As can be seen from the "nodes" requirement, these requirements can be setup as identifiers with the ident argument. A requirement set as an identifier can be used by an interface to select a device with a requirement whose argument matches their passed arguments.

The second part of defining a device plugin is through the implementation. This is where the specific code for each device should be implemented. For the Emotiv headset, this implementation contains all information about how the Emotiv should run. Each device implementation must include a run method which will execute for the device within its own thread. The Emotiv implementation will continually collect data and pass it off through the extension point, which has been forced to be the generic EEG extension due to its implementation as a device.

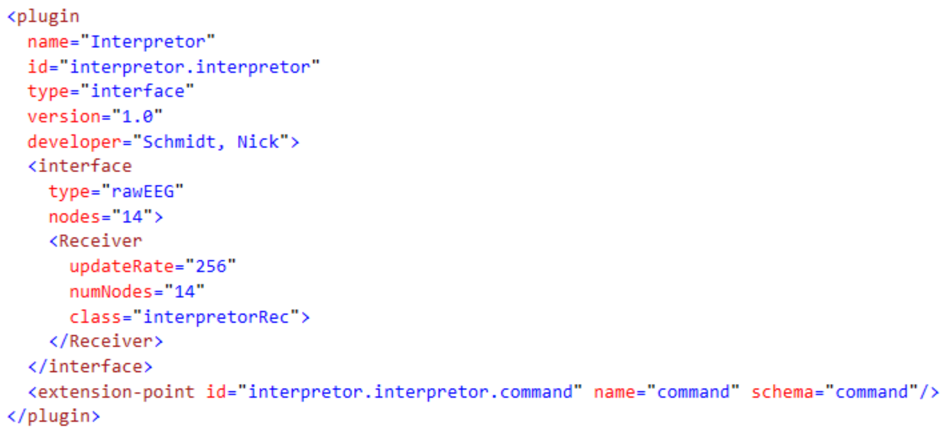


```
<type
  name="Raw EEG"
  type="rawEEG"
  sync="async"
  version="1.0"
  developer="Schmidt, Nick"
  schema="rawEEGExten"
  >
  <requirement name="nodes" type ="int" ident="yes"></requirement>
  <requirement name="updateRate" type="int"></requirement>
</type>
```

Figure 5.3 Device Type Definition

5.5 Interface Plugins

The final type of plugin available is the interface plugin. This type of plugin functions very similarly to the generic plugin. The only difference is that this plugin extends out a device. An example of an interface definition is provided in figure 5.4. This definition file behaves exactly the same as the plugin definition file but requires one extra tag, the interface tag. This tag defines the type of device being accessed, in this case the rawEEG type. Additionally, the interface can pass in any arguments defined as identifiers by the type. In this case, the interface is asking for a rawEEG device implementation which contains 14 nodes. It must also pass in the required receiver element, attributes, and implementation class for its receiver.



```
<plugin
  name="Interpretor"
  id="interpretor.interpretor"
  type="interface"
  version="1.0"
  developer="Schmidt, Nick">
  <interface
    type="rawEEG"
    nodes="14">
    <Receiver
      updateRate="256"
      numNodes="14"
      class="interpretorRec">
    </Receiver>
  </interface>
  <extension-point id="interpretor.interpretor.command" name="command" schema="command"/>
</plugin>
```

Figure 5.4 Interface Definition

5.6 Core Program

The core program within the architecture is responsible for managing all of the plugins, devices, and types within the current implementation. It does so by scanning predefined folders for files of the expected type and format. It begins by scanning for device type implementations, then for device implementations. These devices are given a schema and any defined identifiers are assigned as keys to the possible devices within a type.

After all devices have been loaded, generic plugins and interfaces are scanned. Once these plugins are found, they are parsed and a schema object is generated for each of the extension points within the plugin. These schema objects will be used to ensure that plugins extending an extension point include the correct variables. Any plugin which extends an extension point is added to a list within that extension point for tracking. Additionally, the extension point is added to a list within the extending plugin to allow for two way communication. The core program manages all connections between the plugins by adding them to the relevant plugin and extension point lists based on the IDs provided within the definition file.

After all devices and plugins have been parsed, the run step begins. Each device's implemented run method is called with a new thread. This allows for multiple devices to be driving different sets of plugins. The lists created by the core program then allow for each of the devices to communicate with their interfaces, who communicate with their extending plugins and so on. At the moment these connections are implemented using synchronous connections, meaning that each plugin must finish executing for its extensions before the next plugin can begin. However, the framework and definitions are in place for asynchronous communication with slight changes to threading to allow for the creation of asynchronous communication.

CHAPTER 6. Program Implementation

This chapter will describe the proof of concept program created utilizing this plugin architecture. There are two separate programs for use with different use cases, but these programs could be combined if needed. The first section will describe the training version of the program. The plugins added to this project allow for the recording of EEG data and for the creation of new learning models. The second section will describe the running version of the program. The plugins added to this project allow for the simultaneous usage of the Pacman and lamp exercises.

6.1 Training Program

This section will lay out each of the plugins utilized by the training version of the program. Figure 6.1 shows the overall structure of this plugin for reference during the description.

6.1.1 Device Type: GUI

The first plugin created for use with the training plugin is the GUI type. As described in section 5.4, this device type creates a generic extension point for any device implementing this type. The GUI type includes the guiExten extension point with the following elements: guiBlock, Button, Entry, Text. The guiBlock will hold the Button, Entry, and Text elements. The entry element allows for user input to be processed, the text element allows for text to be placed, and the button element takes a class as an argument which will be called when the button is pressed.

6.1.2 Device Implementation: Training GUI

The implementation of a training type utilized for this project is the training GUI. It is created using TkInter and creates the GUI elements corresponding to any elements passed into it using its

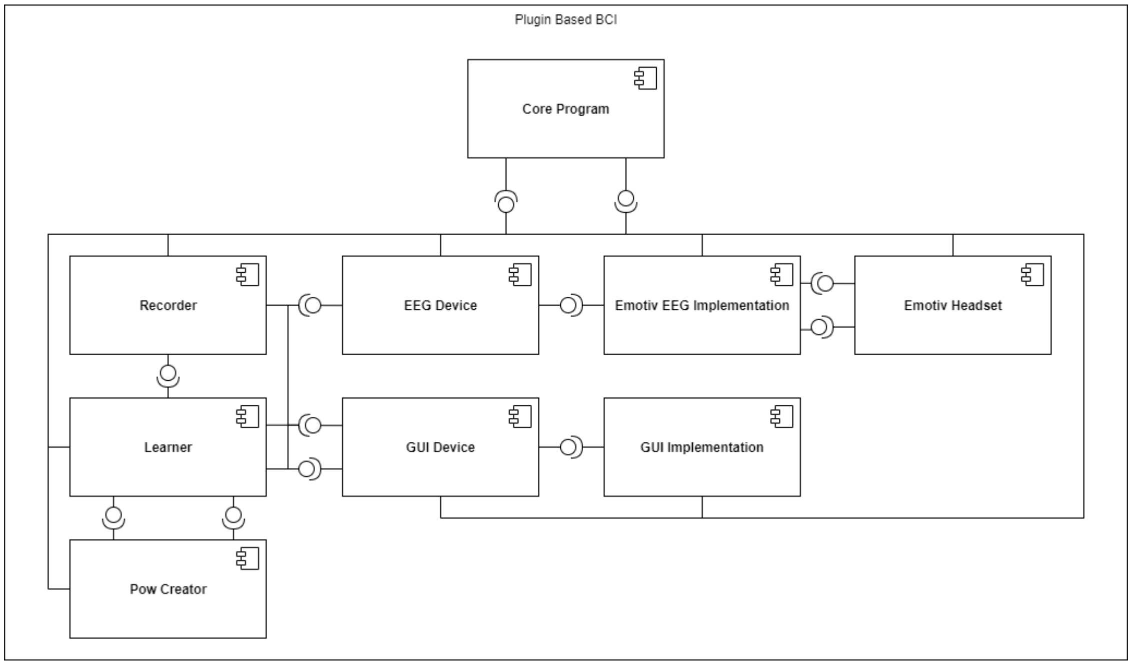


Figure 6.1 Plugin Based Training

GUI extension point. Additionally, it assigns each extending plugin its own subsection of the GUI with its name placed as a divider.

6.1.3 Device Type: EEG

The EEG device type includes a simple extension point which allows for interfaces to register themselves as receivers for the EEG device. It also includes an identifier, numNodes, which allows for an interface to request an EEG device with the number of nodes if it is available. The extension point includes a generic extensionHandler method which will simply pass the received EEG signal to each of the registered extensions. This data should be formatted as a list of raw EEG values with all extra data removed.

6.1.4 Device Implementation: Emotiv

The Emotiv device implements the communication with the Emotiv headset. It begins by registering with the local app by passing it the credentials for login. It then subscribes to the raw

EEG data stream and begins receiving data. Whenever this data is received, it passes it to the EEG extension point so that all extending plugins can access the raw data.

6.1.5 Interface: Recorder

The Recorder interface allows for the collection of raw EEG data into log files. In order to organize these logs, the GUI device type is used. The Recorder plugin registers a GUI Block with two text and two entry fields as well as a button. The text fields are utilized to label the two entry fields. The first entry field contains the name of the model being trained. This name will be used to generate a unique folder for the log files. The second entry field contains the name of the command being recorded. When pressed, the button will generate data for 10 seconds, placing the collected raw EEG file into the folder defined by the first entry field with the name provided by the second entry field. If multiple recordings exist for a single command, a number will be appended to the recording name.

This plugin is also an interface for the EEG device. The recorder plugin listens to the raw EEG data extension point, but only processes and uses the data when a recording is taking place. During this time, the raw EEG data is simply placed into the defined text file. The Emotiv Recorder plugin also contains a single extension point which allows for other plugins to record data without utilizing the GUI.

6.1.6 Plugin: Pow Creator

The PowCreator plugin is utilized for the conversion of raw EEG logs into power band data logs. This plugin contains a single extension point, fileParser, which allows for extending plugins to convert data themselves. This data conversion follows the process described in 3.1 and the output data is saved into separate log files for each command. The data takes the form of a csv file.

6.1.7 Interface: Learner

The Learner plugin implements the machine learning portion of the project. Learner interfaces with the GUI device type to allow for control utilizing the GUI. One text field is used to label the one text entry field present. This entry field allows for the definition of a model name. This model name will be the folder that the Learner checks for power band data. Finally, the GUI extension includes a button which will activate the training method. The model and method used is discussed in section 3.2.2. To verify the data, a quarter of the recordings are set aside for testing. The Learner plugin then creates a model for the given data and outputs it into the provided folder.

The Emotiv Learner plugins extends the Pow Creator plugin. If the data present within the model folder is formatted as raw EEG data, the Emotiv Learner plugin will utilize the Pow Creator plugin for data conversion. The Emotiv Learner plugin also includes it's own extension point, learn, which allows for external plugins to provide a path which the learning algorithm should run on.

6.2 Running Program

This section will lay out the plugins and devices utilized by the running plugin. Figure 6.2 shows the overall structure of this implementation for reference during description.

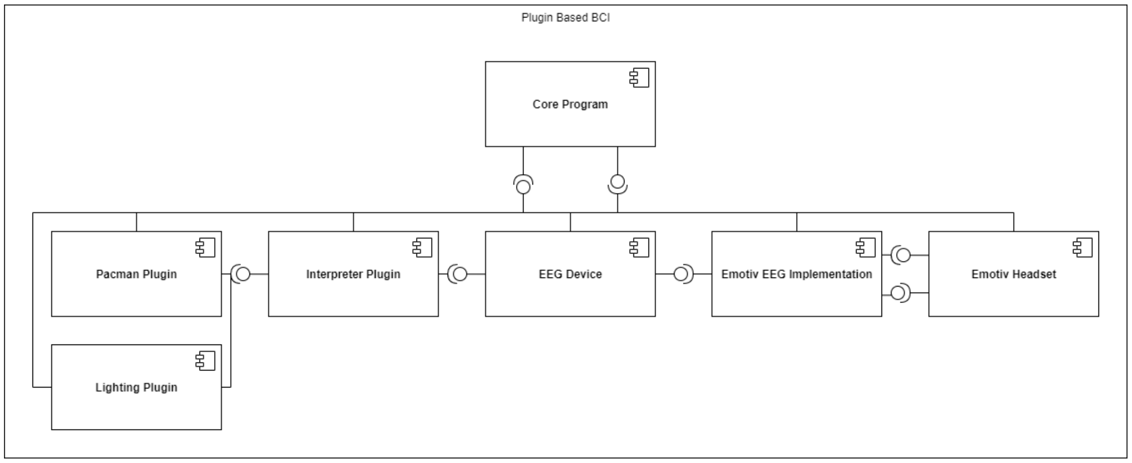


Figure 6.2 Plugin Based Control

6.2.1 Device Type: EEG

The EEG device type includes a simple extension point which allows for interfaces to register themselves as receivers for the EEG device. It also includes an identifier, numNodes, which allows for an interface to request an EEG device with the number of nodes if it is available. The extension point includes a generic extensionHandler method which will simply pass the received EEG signal to each of the registered extensions. This data should be formatted as a list of raw EEG values with all extra data removed.

6.2.2 Device Implementation: Emotiv

The Emotiv device implements the communication with the Emotiv headset. It begins by registering with the local app by passing it the credentials for login. It then subscribes to the raw EEG data stream and begins receiving data. Whenever this data is received, it passes it to the EEG extension point so that all extending plugins can access the raw data.

6.2.3 Interface: Interpreter

The interpreter plugin implements the categorization described in Algorithm 2 within chapter 3. The Interpreter plugin contains one extension points which allows for other plugins to register their desired commands with the interpreter. This plugin interfaces with the EEG device type and collects raw EEG data until 256 samples are collected for categorization. Once this takes place, the FFT data is passed to each extension's handler where the required machine learning model can determine the command that should be run. The extensions are then free to act on this categorization as needed.

The interpreter extension point includes two elements. The first element is the Action Set. This element contains a grouping of commands, each assigned an number index within the machine learning prediction matrix. By default, the command number whose index most closely matches the most recently received prediction matrix will be called. These commands are implemented with a Command Action element. The interpreter extension point allows for the creation of a custom

Action Set class. In this case, the prediction matrix will instead be handled by the user defined class. If no user defined Action Set is used, an extending plugin must implement a Command Action class to define the behavior for each command.

6.2.4 Plugin: Lighting

The lighting program implements the communication with an external Raspberry Pi for control of a light switch. The plugin extends the interpreter plugin, passing a model for use with 2 commands as well as a command to toggle lights. The Interpreter plugin can then handle all of the logic and actions on categorization. The Lighting plugin itself only implements a method to be called with command 1 is activated. This method simply passes the command to toggle to the Raspberry Pi. To do so, the plugin extends the interpreter's extension point, passing the desired method into the Command Action Element within the extension point.

6.2.5 Plugin: Pacman

The Pacman plugin allows for the control of a Pacman simulator using a virtual joystick. The plugin extends the interpreter plugin and provides a model for use with 4 commands. To do so, the plugin passes its own manager into the Action Set element within the interpreter plugin. Rather than relying on the default command handling, the Pacman plugin can now implement its own categorization methodology. The plugin waits for 4 categorizations to take place and then makes a prediction based upon the most common command sent during that time. This command is then passed to the corresponding PyVjoy joystick command to allow for control of the virtual joystick.

CHAPTER 7. Evaluation

This chapter will lay out the evaluation on the accuracy of the utilized machine learning method as well as an evaluation into the successes and failures of the implemented architecture. Where failures or issues are present, theories on fixes or future work are provided.

7.1 Machine Learning Evaluation

Several iterations upon the machine learning methodology were utilized. A sample of tested models is shown in table 7.1. Each data input consists of 4 separate commands. Each command has the same number of data points recorded, with this number being described in table 7.2. Each individual data set consists of a 10 second EEG recording, which is converted into power band data before being utilized by the training model. In order to record test accuracy, three quarters of the data was used for training and one quarter was used in Keras's provided evaluation function.

It should be noted that this data was collected on a user with extensive use of the headset. These accuracy ratings fall on users who do not have experience utilizing the headset. Consistently recreating commands on the headset is an acquired talent. Additionally, the test data accuracy is higher than will be experienced by the user during regular use. This is due to higher irregularities in data during use with other activities as opposed to training. Unfortunately, this data can not be numerically evaluated due to the inability to track a user's desired input in real time.

The second issue with the model training is the variation in readings over time. A few days or even hours after data has been collected, the model begins to fall out of date. This is due to new background noise in the brain patterns as well as variations in headset placement that can cause issues with accuracy.

However, this issue is offset by the fact that it can be seen that relatively small data sets allow for use-able accuracy within the data. Retraining the headset only requires around 4 collections for

Table 7.1 Accuracy Rating of Neural Networks on Datasets

Hidden Layers	Node Count	Model A	Model B	Model C	Model D	Model E	Model F	Model G	Model H	Model I	Model J
1	1	25%	50%	51.19%	73.02%	52.54%	50.79%	72.30%	74.35%	24.96%	32.26%
1	3	80%	100%	57%	100%	100%	98.42%	76.05%	98.81%	99.01%	70.16%
1	7	93.75%	100%	72.23%	100%	100%	100%	98.12%	99.37%	98.30%	95.16%
1	35	97.19%	100%	73.41%	100%	100%	100%	98%	99.32%	99.29%	95%
1	50	97.80%	100%	72.22%	100%	100%	100%	99%	99.39%	98.17%	91.19%
2	30, 7	97.96%	100%	72.62%	100%	100%	100%	95%	98.81%	98.73%	93.55%
2	50, 13	97.97%	100%	73.41%	100%	100%	100%	97.02%	99.39%	99.81%	94.35%

Table 7.2 Dataset Sizes

	Model A	Model B	Model C	Model D	Model E	Model F	Model G	Model H	Model I	Model J
Data Points	10	5	4	4	5	5	10	30	11	2

each command. This means that if a user will be having a session with the headset they will only need to record 160 seconds of data (4 commands * 4 collections * 10 seconds a collection). Finally, these accuracy issues do not apply with models utilizing two commands.

Overall, this learning method is effective. It can be observed in table 7.1 that a user is capable of issuing commands to software or a device with a high level of accuracy. When compared to the Emotiv Epoc+'s built in categorization method, the model is at least as effective. The Emotiv methodology also suffers from the same drawbacks, requiring a daily re-calibration as well as suffering from accuracy issues when more commands are added. It can be observed from the collected data that this is true of the Keras neural network method, but support for 4 commands is also achievable by an experienced user.

7.2 Architecture Evaluation

In order to determine whether an EEG device could be utilized to control real world objects, an architecture that could support the communication with these devices would have to be established. For this architecture to be as effective as possible, it should be capable of controlling more than one device, and should allow for modification to support new devices and software in order to resolve the issues encountered by other implementations. To evaluate the effectiveness of the architecture, several criteria were utilized: modifiability, usability, and scalability.

In order to measure modifiability, a few requirements were laid out for the architecture. First, a new plugin should not require knowledge of another plugin's implementation. This ensures that new plugins can easily be developed for new devices without complex knowledge about any step of the process. Second, a new plugin should be added to the core functionality of a program without requiring user input. This ensures that new devices and software can be supported without having to modify the existing functionality of the program. Third, a new plugin should be able to allow other plugins to extend it. This will allow for devices to be accessed by other devices and greatly increase the functionality of the EEG communication.

It can be shown that the current architecture meets these modifiability requirements. This was accomplished in several iterations, with early iterations on the architecture failing to meet the necessary requirements. First, a new plugin does not require knowledge of another plugin's implementation. This is demonstrated in Chapter 6, the only necessary knowledge for implementation is the format and meaning of the extension points. Second, a new plugin can be added to the core functionality of the program without requiring user input. To add a plugin to the system, it must be added to the plugin folder. After this is complete, the plugin will be automatically added to the system. Third, a new plugin can add its own extension points as well as data format. This is accomplished through the xml definition files laid out in the architecture as well as the python implementation files and extension point files.

In order for the plugin to meet the usability requirement, it should not require user knowledge of plugins or BCI devices to use. This is one of the issues identified with the OpenVIBE solution where users must create their own BCI logic flow. This is accomplished through the implementation of BCI devices as well as the nature of a plugin architecture. This ensures that new devices and software can be added to the system through pre-developed plugins. A user would not need to have an understanding of how a BCI would work, they would just need to grab a set of plugins which accomplish all of this for them.

The final criteria for the success of the system is scalability. The architecture should not suffer when more plugins are added to the system. Currently, this issue is partially addressed. In the final iteration of the architecture, devices are each provided with a thread to run in. This allows for a larger level of scalability between devices. Additionally, a skeleton framework has been implemented to allow for asynchronous execution of plugins. However, at the moment this has not been fully implemented. This means that a plugin which sits idle could slow down the other plugins as they wait for their turn with the data. Further work must be completed to fully implement the asynchronous execution of plugins and solve scalability issues.

The architecture overall fits the needs of a BCI architecture and solves the compatibility and usability issues identified in other solutions. It allows for the easy extension of the core BCI framework and allows for the use of multiple type of BCI device. It implements a GUI as well as the full implementation of plugins for the execution of commands for an EEG device. This allows for the development of new device controllers with no machine learning or BCI knowledge.

CHAPTER 8. Conclusion

It can be seen from the evaluation of the architecture as well as the machine learning model that the question: ”Can a user control a set of digital objects using their brainwaves?” has been answered positively. In the current architectural implementation, a user is capable of controlling multiple digital and physical devices simultaneously with an EEG device. This device reads in brainwave data, and passes it to a machine learning algorithm which can accurately categorize these commands. The implemented architecture allows for the extension of these core functionalities to include any new devices. This is accomplished through robust custom interface definitions.

However, it can also be determined that future work can be accomplished with this solution. First, more experimentation and data collection can be accomplished on new users. This will allow for a more universal and in depth machine learning algorithm. Second, the architecture should be extended to allow for asynchronous execution of plugins. This will ensure that the architecture is scalable to as many plugins as possible, with slower plugins consuming data as they finish execution. Finally, more methods of BCI can be tested with the architecture. The architecture is implemented to allow for the implementation of new BCI methods, but none were available for testing. These areas of future research would allow for further development of a universal BCI platform that can be utilized for the control of multiple physical and digital objects.

Bibliography

[1] Emotiv: Emotiv pro

[2] Wikipedia: Artificial neural network (2020)

[3] Sharjith: Simple plug-in architecture in plain c (2012)

[4] Bogue, R.: Brain-computer interfaces: control by thought. Industrial Robot **37**(2) (2010) 126–132

[5] Shih, J., Krusienski, D., Wolpaw, J.: Brain-computer interfaces in medicine. In: Mayo Clinic Proceedings, Mayo Clinic (2012) 268–279

[6] Waldert, S.: Invasive vs. non-invasive neuronal signals for brain-machine interfaces: Will one prevail? Frontiers in Neuroscience **10**(295) (2016)

[7] Kamp, A., Pfurtscheller, G., Edlinger, G., da Silva, F.L.: Electroencephalograph: Basic Principles, Clinical Applications, and Related Fields. Ippincott Williams & Wilkins, Philadelphia, PA (2005)

[8] Vourvopoulos, A., Liarokapis, F.: Robot navigation using brain-computer interfaces. In: 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications. (2012) 1785–1792

[9] Baars, B., Gage, N.: Cognition, Brain, and Consciousness: Introduction to Cognitive Neuroscience. 2, revised edn. Academic Press, Cambridge, MA (2010)

[10] Zhang, W., Tan, C., Sun, F., Wu, H., Zhang, B.: A review of eeg-based brain-computer interface systems design. Brain Science Advances **4**(2) (2018) 156–167

[11] da Silva], F.L.: Neural mechanisms underlying brain waves: from neural membranes to networks. Electroencephalography and Clinical Neurophysiology **79**(2) (1991) 81 – 93

[12] Koudelková, Z., Strmiska, M.: Introduction to the identification of brain waves based on their frequency. MATEC Web of Conferences **210** (01 2018) 05012

[13] Al-Fahoum, A., Al-Fraihat, A.: Methods of eeg signal features extraction using linear analysis in frequency and time-frequency domains. ISRN neuroscience **2014** (02 2014) 730218

[14] Shahid, N., Rappon, T., Berta, W.: Applications of artificial neural networks in health care organizational decision-making: A scoping review. PLOS ONE **14**(2) (02 2019) 1–22

[15] Chollet, F.: Keras

[16] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016) `http://www.deeplearningbook.org`.

[17] Birsan, D.: On plug-ins and extensible architectures. ACM Queue **3** (2005) 40–46

[18] WordPress: Wordpress

[19] Bolour, A.: Notes on the eclipse plug-in architecture (2003)

Appendix A

Configuration: Running

1. Plugin: EEG Type
   (a) Description: Describes the implementation of EEG device types. Allows for generic interface.
   (b) Requirements
      i. Nodes
      ii. Updates per Second
   (c) Extension Point: EEG
      i. Receiver
         A. Update Rate
         B. Number of EEG Nodes
2. Plugin: Interpreter
   (a) Description: Converts raw EEG voltages into power band data and outputs which command is being input.
   (b) Interface: Interfaces with a Raw EEG device.
   (c) Extension Point: Command
      i. Description: Extensions to this point can define a set of commands, interpreter will output which command is being input.
      ii. Element: Action Set
         A. Description: Set of commands that can be input
         B. setName: Name of the command set
         C. setDescription: Description of the command set
         D. setModel: Path to the Keras model that should be used for categorizing the power band input

E. setWeights: Path to the weights for the Keras model that should be used for categorzing the power band input

F. class: class that implements full actionSet categorization, command classes won't be used if this is used. Action set will handle command selectoin

iii. Element: Command Action

A. Description: Describes the command that should be activated when this command is input

B. actionName: Name of the command

C. actionDescription: Description of the command

D. commandNumber: Index number of this command, should be from 0-N

E. commandTimer: Amount of time that must pass before this command can be used again, can be 0

F. class: Class that implements command action if wanted

3. Plugin: Lighting

(a) Description: Communicates with a webserver running on a Raspberry PI to toggle a light switch

(b) Extends: Extends Interpreter.Command

i. setName: Light Controls

ii. setDescription: A set of actions to control a light switch through a connection to a raspberry PI

iii. setModel: Ligthing.json

iv. setWeights: Lighting.h5

v. commandAction

A. actionName: Toggle Light

B. actionDescription: Toggle the light using the PI

C. commandNumber: 1

D. commandTimer: 5

E. class: lightToggle

4. Plugin: Pacman

   (a) Description: Communicates with a webserver running on a Raspberry PI to toggle a light switch

   (b) Extends: Extends Interpreter.Command

      i. setName: Joystick Controller

      ii. setDescription: A set of actions to send command to vJoy a virtual joystick emulator

      iii. setModel: Pacman.json

      iv. setWeights: Pacman.h5

      v. class: joystickManager

Configuration: Training

1. Plugin: EEG Type

   (a) Description: Describes the implementation of EEG device types. Allows for generic interface.

   (b) Requirements

      i. Nodes

      ii. Updates per Second

   (c) Extension Point: EEG

      i. Receiver

         A. Update Rate

         B. Number of EEG Nodes

2. Plugin: Device Type

   (a) Description: Describes the implementation of GUI devices
   (b) Extension Point: GUI
      i. Element: GUI Block
         A. Button: Button to activate a function
         B. Text: Text to display on the GUI
         C. Entry: Text Entry form
         D. Name: Name of the GUI element
         E. Description: Description of the GUI Element
      ii. Element: Text
         A. Text: Text to display
         B. Width: Width of the element
         C. Height: Height of the element
      iii. Element: Button
         A. Text: Text to display on the button
         B. Class: Class to activate when button is pressed (path to .py File)
      iv. Element: Entry
         A. ID: ID that can be referred to in other elements to access the text in this entry form

3. Plugin: PowCreator
   (a) Description: Converts EEG data logs into power band data logs
   (b) Extension Point: FileParser
      i. Element: ParseHandler
         A. readPath: Path that the EEG logs are in
         B. outPath: Path that the power band logs should be placed into

C. recordLength: Amount of EEG records that should be read in before a power band calculation is performed

D. incrementAmount: Amount of old elements that should be dropped after a power band calculation is performed

4. Plugin: Emotiv Learner
   (a) Description: Reads log files categorized by command to crates a Keras model for the data
   (b) Interface: GUI
      i. Name: Emotiv Learn
      ii. Description: Learn the logs currently in the log folder
      iii. Element: Text
         A. Text: Model Name
         B. Width: 15
         C. Height: 1
      iv. Element: Entry
         A. ID: learnModelName
      v. Element: Button
         A. Text: Learn Model
         B. Class: learnButton
   (c) Extends: powCreator.powCreator.fileParser
      i. parseHandler
         A. readPath:
         B. outPath:
         C. recordLenth: 256
         D. incrementAmount: 16

(d) Extension Point: Learn

    i. Elements

        A. Element: LearnManager

        B. modelPath: Path to the model

        C. modelName: Name of the model

5. Plugin: Emotiv Recorder

    (a) Description: Records data from a EEG Headset.

    (b) Interface: GUI

        i. Name: Emotiv Recorder

        ii. Description: Learn the logs currently in the log folder

        iii. Element: Text

            A. Text: Model Name

            B. Width: 15

            C. Height: 1

        iv. Entry

            A. ID: recorderModelName

        v. Element: Text

            A. Text: Command Name

            B. Width: 15

            C. Height: 1

        vi. Element: Entry

            A. ID: recorderCommandName

        vii. Element: Button

            A. Text: Record Log

            B. Class: recordButton

(c) Interface: EEG

(d) Extension Point: recordLog

    i. Element: LogRequest

        A. logName: Name of the log file that should be used

        B. commandName: Name of the command being used

        C. logTimer: Amount of time in seconds the log should be recorded for