CEG 4110
P2: Wearable Epilepsy Device & MC Team
T4: Patterns Matching, Association & Prediction

AUTHORS

IAN S. BARNEY
COMPUTER SCIENCE MAJOR
WRIGHT STATE UNIVERSITY

ELLIOT C. FRIES
COMPUTER SCIENCE MAJOR
WRIGHT STATE UNIVERSITY

JAMES S. LANGUIRAND
COMPUTER SCIENCE MAJOR
WRIGHT STATE UNIVERSITY

CLAYTON D. TERRILL
COMPUTER ENGINEERING MAJOR
WRIGHT STATE UNIVERSITY

CONTENTS

1 ABSTRACT

The authors of this report were tasked with the Patterns Matching, Association & Prediction of the Wearable Epilepsy Device & MC Team. They implemented the Unified Process which contains five workflows. The workflows include Requirements, Analysis, Design, Implementation, and Testing. The requirements for the entire project were determined first. This set the outlying blueprint for every other workflow. Using the requirements, a contract was drawn up. Then, during the Analysis workflow, nouns were extracted from the requirements summary. These nouns were used to determine the classes and attributes needed for the program. The Design workflow then mapped out the data flow necessary to implement the classes. Once the analysis and design were completed, the implementation workflow began. The implementation workflow was when the code was produced following the guidelines set forth by the analysis and design. When the program was finished, the testing workflow determined if the program met the requirements without any errors. Even though the original program did, the authors were not finished. They created a UML diagram representing their program and then compared it with the team before them. During this step, it was discovered that there was major overlap. So, the authors modified their UML and requirements. This created a moving target problem which required the authors to redo every workflow. Once every workflow was made to accommodate the new requirements, they successfully fixed the moving target problem. The final program had been produced and it was ready to be integrated into the Wearable Epilepsy Device.

2 INTRODUCTION

Our team was tasked with the Patterns Matching, Association & Prediction of the Wearable Epilepsy Device & MC Team. To complete our portion of the project we implemented the Unified Process. The Unified Process abides by the five core workflows. The five core workflows include Requirements, Analysis, Design, Implementation, and Testing. Our team consisted of five members who each were assigned one of these workflows. However, the member assigned to do the testing dropped out of the team. Consequently, us remaining four had to complete the testing.

The paradigm applied alongside the Unified Process was the object-oriented paradigm. Using the object-oriented paradigm, we completed each workflow iteratively and incrementally. First, we created a contract that outlined all the requirements of our portion of the project. Then, we analyzed the requirements and derived noun classes. Using these noun classes, we designed a top-down decomposition of how the data must flow between the classes. Now, we had a thorough understanding of what our portion of the project must accomplish. So, we implemented the classes and flow of data. To ensure that our portion of the project behaved properly, we thoroughly tested it using various testing methods. Using the Unified Process, we successfully fashioned our portion of the project. Yet, we ran into one of the biggest issues that could have occurred during the UML stage. Our portion of the project was not compatible with the team before us.

Since we were the final team within the Wearable Epilepsy Device project, our job was to receive five signals from the previous team and predict if an epileptic seizure was occurring. Originally, we were instructed that we would be detecting the abnormalities in signals and then match the abnormality pattern with issues that cause epileptic seizures. However, when conferring with the team sending us signals, we discovered that our UMLs did not match up. They were detecting patterns and sending out Boolean values representing whether a signal was abnormal or not. Thus, we had to modify our requirements and a moving target problem was born.

To fix the moving target problem we decided to modify the artifacts created during each workflow. We no longer needed to detect abnormalities. All we had to do now was to compare the abnormality patterns with the issue patterns. The reducing of our requirements really simplified our portion of the project. We were able to simplify our requirements overview, reduce what classes had to accomplish, and reduce the amount of data flow needed. The implementation and the testing of the program were also simplified as a result.

During this report, we are presenting the original and modified versions of the workflow artifacts. We will first show the original workflows. Then, we will illustrate our final workflows and the modifications made to them. This report presents the steps we took to reach our final portion of the project that could be successfully integrated with the other team's portion of the project.

3 ORIGINAL PROJECT

3.1 REQUIREMENTS

Requirements Supervisor: Clayton D. Terrill

3.1.1 Overview

The first step of the Unified Process was to determine the requirements. The purpose of the requirements workflow was to understand the application domain and determine the customer's needs. Therefore, we listened to the overall goal of the Epilepsy Device to reach an overall understanding of the project. Then, we talked one on one with the customer and discovered what was wanted for our portion of the project.   We sifted through these wants and determined what the customer needed. Thus, the requirements document, or contract, was born.

The first aspect of the original requirements contract was to define exactly what our part of the project needed to do. We were told that we would be receiving five signals representing integer values. These signals would then be compared to a corresponding statistics regarding the patient. These statistics would represent the normal range that each signal should be in. If the signal was outside of this range, then it was deemed abnormal. Following the detection of abnormalities, we were required to take the pattern of the abnormal signals and compare it with patterns that represent a specific issue. Each issue is associated with an epileptic seizure. So, if an issue pattern matched an abnormal signal pattern, then we outputted the issue.

Once the application domain was understood and the outline of the project was completed, we created a contract and assigned jobs for each workflow. The first workflow was the Requirements workflow which is completed by Clayton D. Terrill. Then, the next workflow is the Analysis workflow which was completed by Elliot C. Fries. Next, the Design workflow was completed by James S. Languirand. Ian S. Barney used the previous workflows as reference to complete the Implementation workflow. Lastly, the Testing workflow was assigned to Jeremy M. Watson. However, Watson did not complete his workflow or participate in the team. So, the team leader, Clayton D. Terrill, completed this workflow.

3.1.2 Contract

Abnormalities within a person's life signals may warn of an upcoming epileptic seizure. So, there will be five signals that the frequency will be monitored for. The signal frequencies will be compared to the patient's normal statistics to determine if there is an abnormality. Abnormalities within the signals may match a pattern that is associated with an issue. If an issue is detected, a message for the epileptic seizure will be outputted.

Team Member Requirements:

- All team members are required to attend every workshop class and is responsible for their assigned job.
- The project will be written in C++.
- The project will be written from scratch by the team members.
- The team leader will make sure that all the workflows are completed accordingly.
- Team members will write a report regarding the project.
- Team members will create a PowerPoint to present.
- The due date is April 17th, 2018.

Names	**Jobs**	**Status**
Clayton Terrill	Requirements and Team Leader	**Active**
Elliot Fries	Analysis	**Active**
James Languirand	Design	**Active**
Ian Barney	Implementation	**Active**
Jeremy Watson	Testing	**Missing**

3.2 ANALYSIS

Analysis Supervisor: Elliot C. Fries

3.2.1 Overview

The analysis workflow was used to determine what classes were needed for our portion of the project. In our original analysis, we had eight nouns that could be classes or attributes. These nouns included Abnormalities, Signal, Seizure, Frequency, Statistics, Pattern, Patient, and Issue. The nouns were derived from the requirements to determine the classes and attributes needed for a successful design and implementation. Based on the nouns, we chose three classes to use. The three classes included Signal, Issue, and Patient.

The Signal class was created to handle the incoming signals and store the frequency they carry. The Signal class was meant to have multiple instances and continuously read from incoming signals. Therefore, the only purpose of the Signal class was to act as a boundary class and receive input from an outside source.

The Issue class held a pattern and a message to describe it. The pattern represented what signals must be abnormal for the issue to occur. If an issue occurred, then a meaningful message describing the issue must have accommodated it. Therefore, it was necessary to store a message within the Issue class. The Issue class was exclusively an entity class which contained a pattern and message associated with an issue.

The purpose of the Patient class was to store the patient statistics and compare them with the five signal frequencies. The patient statistics contained the normal frequency ranges for each signal. If a certain range was exceeded by a signal, then the signal was deemed abnormal. The pattern of abnormal signals were then compared with patterns associated with an issue. Therefore, the Patient class was both an entity class and control class. The entity portion stored the patient statistics and whether a signal was abnormal. Meanwhile, the control portion compared the patient statistics with the incoming signals, updated whether the signals were abnormal, and compared the abnormal signal patter with the issue patterns. If an issue matched the pattern of abnormal signals, then the issue would be returned. In hindsight, combining the control and entity portions of the Patient class negatively affected the cohesion within the class. It would have been best to separate the entity and control portions into two distinct classes.

3.2.2 Noun Class Extraction

Abnormalities within a **patient's life signals** may warn of an upcoming epileptic **seizure**. So, there will be five **signals** that the **frequency** will be monitored for. The **signal frequencies** will be compared to the **patient's** normal **statistics** to determine if there is an **abnormality**. **Abnormalities** within the **signals** may match a **pattern** that is associated with an **issue**. If an **issue** is detected, a **message** for the epileptic **seizure** will be outputted.

- Nouns:
    - **Abnormalities, Signal**, **Seizure, Frequency, Statistics, Pattern, Patient**, **Issue**
        - **Abnormalities** is an abstract noun which is associated with a patient. Abnormalities are associated with patterns within the Patient class. Therefore, use abnormalities as an attribute within the Patient class.
        - **Frequency** is an abstract noun which is associated with a signal. Frequency represents the integer value being received in the Signal class. Therefore, use frequency as an attribute within the Signal class.
        - **Statistics** is an abstract noun which is associated with a patient. Statistics represents the normal frequency values for a patient's vital signals. Therefore, use statistics as an attribute within the Patient class.
        - **Seizure** is an abstract noun which is associated with issues. Issues describes what is causing the seizure. Therefore, a seizure class or attribute is not needed.
        - **Pattern** is an abstract noun which is associated with an issue. Each issue has a pattern of abnormalities associated with it. Therefore, use pattern as an attribute for the issue class.
        - **Message** is an abstract noun which is associated with an issue. Each issue needs to be identified. Therefore, use message as an attribute for the issue class.
- Patient Class
    - Description
        - Creates object representing the patient.
    - Entity Class and Control Class
        - Holds the normal stats for the patient and compares them with the signal.
    - Attributes
        - Abnormalities – Represents if an abnormality was detected within a signal.
        - Statistics – Normal signal frequencies for the patient.
- Signal Class
    - Description
        - Creates objects representing the Integer values being received from the signals.
    - Boundary Class
        - Receives input from an outside source.
    - Attributes
        - Frequency – The frequency being received from the signal.
- Issue Class
    - Description
        - Creates objects representing the issue patterns.
    - Entity Class
        - Holds the pattern of abnormalities representing the issue.
    - Attributes
        - Pattern – Represents what pattern of signals causes the issue.
        - Issue Message – The output describing what issue is occurring.

3.3 DESIGN

Design Supervisor: James S. Languirand

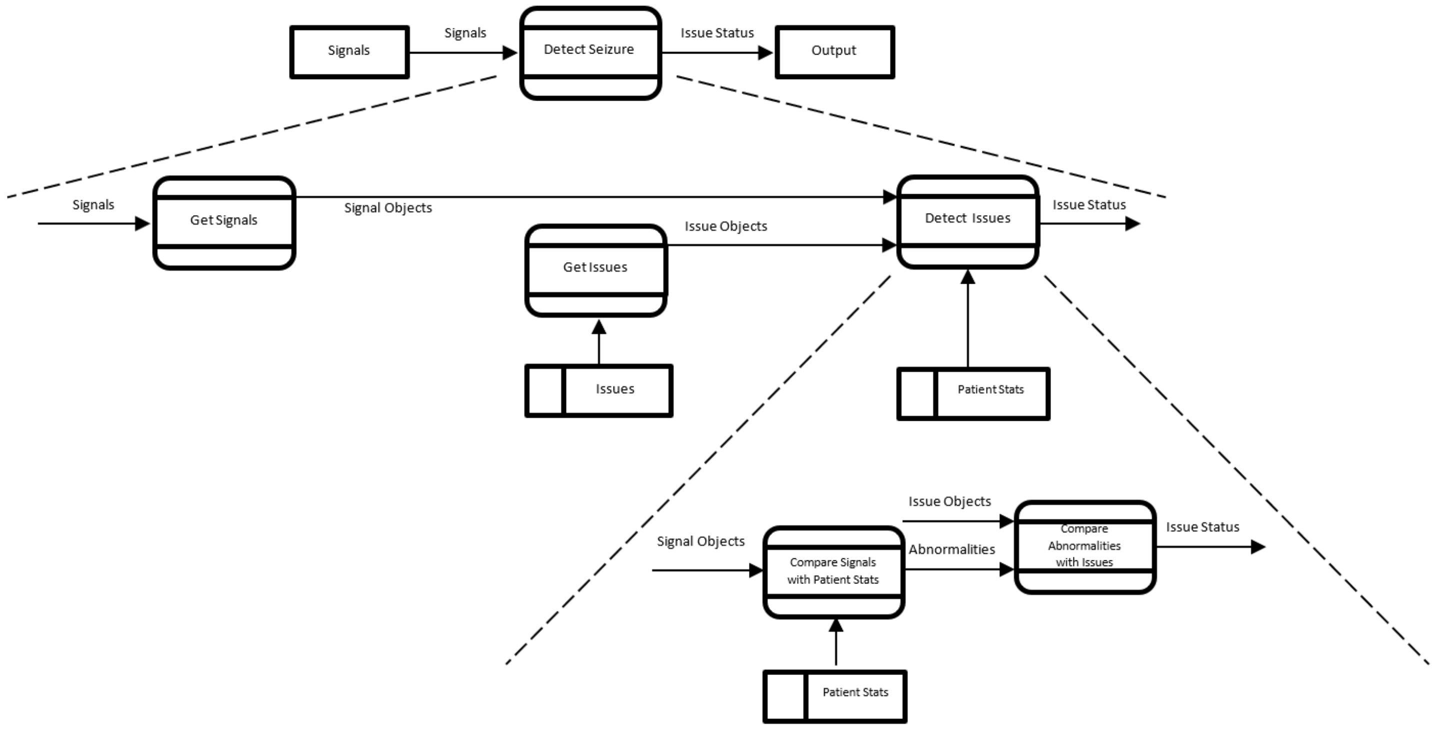
3.3.1 Overview

The Design workflow was used to model the flow of data using a top-down decomposition diagram. The design models the flow of data and provides a blueprint for an implementor to use to successfully implement a program. Thus, we decomposed each part to reach their smallest possible level. The top level of decomposition consisted of signals being received, detecting an issue, and then outputting the issue. The top level was an overview of what the program does and was decomposed even farther.

The second level of decomposition outlined what needed to be created and loaded to detect an issue. This level converted signals to objects while also loading the issue objects. The signal objects held the frequency integer values representing the signal. The signal objects were grouped into an array and passed to the Patient class. Meanwhile, the issue objects were created on startup using input from a file or user. These objects contained a pattern of abnormalities and a message representing the issue. The issue objects were also grouped into an array and passed into the Patient class. The Patient class is where seizures are detected and acts as the next level of decomposition.

The third level of decomposition delves into how the program will detect if an issue has occurred. The signal objects were compared with the patient statistics normal ranges to detect abnormalities. The patterns of abnormalities that were produced by the signals would then be compared with the issue objects. If the pattern of abnormalities matched an issue, the issue would be outputted. This was the lowest we could decompose our program. Therefore, the design was ready to be sent to the implementation workflow.

3.3.2 Top-Down Decomposition Data Flow Diagram



3.4 IMPLEMENTATION

Implementation Supervisor: Ian S. Barney

3.4.1 Overview

The implementation workflow was where the program was produced. During the implementation process, the implementor was the only one who had access to the source code. The classes outlined in the analysis were created at first. Then, attributes were added to each class as variables. Each class was given setters and getters to access and modify the attributes. Then, the control methods were implemented. The three classes for the program created a flow of data that was outlined by the design diagram. The three classes implemented were Signal, Issue, and Patient.

The Signal class acted as a Boundary class for the program. Therefore, it was necessary to create setters and getters for the variables. The only variable in the Signal class was called frequency. Frequency represented the integer value of an incoming signal. This incoming signal constantly updated the frequencies in the Signal class objects. Since there were five signals being received, this Signal class produced five objects that held a frequency within them.

The Issue class acted as an Entity class as it held information regarding a specific issue. So, a constructor that allowed the variables to be set was implemented. The variables used included a message and a Boolean pattern. The message variable represented what the issue is. The purpose behind the message was to display what was causing and epileptic seizure. The pattern variable represented the pattern of abnormalities that are associated with the issue. There was no limit on how many Issue objects that could be created. If two objects shared a pattern, then our implementation would only display the first one which was created. If an Issue needed to display a different message or use a different pattern, then the setters provided could be used to modify the variables. The getters allowed access to the variables from the control portion of the Patient class.

The Patient class is both an Entity class and a Control class that is used to detect if signal abnormalities match and issue pattern. The statistics variable is being used as a 2D array to store the normal ranges for each signal. The two rows represented the lower and upper bounds of the normal ranges while the columns represented a certain signal. The stats could be assigned on creation or modified using a setter method. A control method was needed to check the ranges with the signal objects to see if any are out of the normal ranges. If the signal object's frequency was out of the normal range of that signal, then that signal was set as abnormal within an abnormal Boolean array. The Boolean array is then compared with all the Issue objects using another control method. If the pattern of an Issue object matches the abnormal Boolean array, then that Issue's message is returned. The returned message indicates the Issue that is causing an epileptic seizure.

3.4.2 Code Documentation

3.4.2.1 Signal.h

- Overview
  - Declares the variables and function signatures used in the Signal.cpp.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Variables
  - frequency
    - Integer value that represents the incoming frequency.
- Function Signatures
  - Signal();
  - Signal(int);
  - ~Signal();
  - void setFrequency(int);
  - int getFrequency();

3.4.2.2 Signal.cpp

- Overview
  - The Signal Class is used to create a Signal object that monitors signals being received. The Signal Object's Integer value will be constantly changed to represent if the Signals frequency.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Methods:
  - Signal()
    - Default Constructor of the Signal. Initializes the frequency to be 0.
  - Signal(int frequency)
    - Constructor for when a value has been designated during Signal object creation. Sets the frequency.
    - @param frequency - Integer value to set frequency with.
  - ~Signal()
    - Default Destructor that deletes the Signal. Prevents Memory Leak.
  - setFrequency(int frequency)
    - Sets the frequency variable with an Integer value.
    - @param frequency - Integer value to set frequency with.
  - getFrequency()
    - Returns the frequency of the Signal.
    - @return frequency - Integer value for the frequency.

3.4.2.3 Issue.h

- Overview
  - Declares the variables and function signatures used in the Issue.cpp.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Variables
  - message
    - String value that displays the Issue.
  - pattern[5]
    - Boolean array that represents the Issue Pattern.
- Function Signatures
  - Issue();
  - Issue(string message, bool pattern[]);
  - ~Issue();
  - void setMessage(string message);
  - void setPattern(bool pattern[]);
  - bool* getPattern();
  - string getMessage();

3.4.2.4 Issue.cpp

- Overview:
  - Issue Class is used to create an Issue object that represents an Issue Pattern. The Issue Pattern will be compared with signals to see if that specific Issue is occurring.
- Authors:
  - Clayton D. Terrill and Ian S. Barney
- Methods:
  - Issue()
    - Default Constructor of the Issue. Initializes the pattern to be all true. Initializes the message as 'Detected an Issue'.
  - Issue(string message, bool pattern[]);
    - Constructor for when values have been designated during Issue object creation.Sets the message for the Issue.Sets the pattern for the Issue.
    - @param message - String to set the message with.
    - @param pattern[] - Boolean array to set the pattern with.
  - ~ Issue()
    - Default Destructor that deletes the Issue. Prevents Memory Leak.
  - setMessage(string message)
    - Sets the message variable with a String value.
    - @param message - String to set the message with.
  - setPattern(bool pattern[])
    - Sets the pattern array.
    - @param pattern - The array to set the pattern array with.
  - getPattern()
    - Returns Issue Pattern.
    - @return bool* - Boolean array to set the pattern with.

  - getMessage()
    - Returns the Issue message.
    - @return string - String containing the Issue message.

3.4.2.5 Patient.h

- Overview
  - Declares the variables and function signatures used in the Patient.cpp.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Variables
  - stats[2][5]
    - 2D Integer Array that represents the normal ranges for the patient.
  - isAbnormal [5]
    - Boolean Array that represents the pattern of abnormal signals.
- Function Signatures
  - Patient();
  - Patient(int stats[][5]);
  - ~Patient();
  - void setStats(int stats[][5]);
  - int getStats(int, int);
  - void checkRanges(Signal signals[]);
  - string matchPattern(Issue issues[]);

3.4.2.6 Patient.cpp

- Overview
  - Patient Class is used to create an Patient object that represents a Patient. The Patient will store the normal ranges for each signal as stats. These stats are compared to the signal values and determines if the signals or abnormal or not. Abnormalities are stored in an array. The abnormal array is then compared with Issues to see if the abnormal signals match an Issue pattern that denotes a seizure.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Methods:
  - Patient ()
    - Default Constructor of the Patient. Initializes the Patient stats to be all 0. Initializes the isAbnormal array to be all false.
  - Patient(int stats[][5])
    - Constructor for when attibutes are also passed in during the Patient's creation. Sets the Patient stats array. Sets the isAbnormal array all to false.
    - @param stats - The 2D Intege Array of provided stats.
  - ~Signal()
    - Default Destructor that deletes the Patient. Prevents Memory Leak.
  - setStats(int stats[][5])
    - Sets the Patient stats array with the provided stats.
    - @param stats - The array of provided stats.

- getStats(int row, int col)
    - Returns the specified stat at a certain row and column.
    - @param row - The index of the desired row.
    - @param col - The index of the desired column.
    - @return int - Integer containing the specified stat.
- checkRanges(Signal signals[])
    - Checks to see if the Signal frequencies are within the Patient ranges. Sets corresponding Boolean values within the isAbnormal array.
    - @param signals - The Signal array to be compared with.
- matchPattern(Issue issues[])
    - Compares the Signals in the Signal array with the isAbnormal pattern.
    - @param issues - The Issue array to be compared with.
    - @returns – Returns message String representing Issue status.

3.4.3 Online Code Link

- The code and project artifacts were uploaded to GitHub for easy access.
    - https://github.com/Terrillc13/PatternsMatchingAssociationPrediction/tree/OriginalProject
- The code and project artifacts will also be included in the .zip file.

3.5 TESTING

Testing Supervisor: Clayton D. Terrill

3.5.1 Overview

The most difficult workflow was by far the Testing workflow. Testing was used to verify that every workflow was properly completed to produce a valid product. The product must fulfill the requirements. Likewise, the functionality for each class and the overall program needed to be assured. So, we tested each class path using White-Box Testing. To perform the testing, we created a test harness in place of the main class.

The test harness carried out all the tests we needed to complete. First, we needed to make sure each class worked as expected. Each class would have to be tested to assure that the values in them behaved as expected. So, we used Unit Testing. The class variables were all tested with setters and getters for each class.

The Signal class needed to be able to constantly update its frequency value using a setter. Then, the control class needed to read the Signal's frequency. So, the getter was tested to assure it returns the appropriate value. The Issue class needed to be able to hold a pattern and message. Thus, the functions to set the Issue variables were checked. Then, the control class had to read the pattern and message from the Issue class. So, the getter for both variables were also tested. Lastly, the Patient class needed to be tested. The Patient class was tested last because it was data dependent on the other two classes. White-Box Testing was used within the Patient class because it needed valid ranges and used detection algorithms.

The White-Box Testing performed on the Patient class was quite extensive. We had to ensure that the Patient Stat Ranges entered were valid. Patient Stat Ranges were only valid if the first number is followed by a larger number. So, we had to test that a range would not be set where the larger number came first. Then, we had to makes sure that the abnormality detection was working. Boundaries needed to be tested for the Patient Stat Ranges. The ranges needed to be tested for if a signal frequency was exactly in the ranges or one out of it. Lastly, the Issue pattern matching needed testing. We created three test issues and inputted signals that would produce them. Using these test, we were able to prove that our program worked without any errors.

3.5.2 Tests

3.5.2.1 Test Harness

- Created a test harness in the main to perform every test
  - Default Test Values Used for Various Tests
    - Issue 1
      - Pattern
        - True, True, True, True, True
      - Message
        - Seizure Detected: EMERGENCY (Complete bodily shutdown)
    - Issue 2
      - Pattern
        - True, False, False, False, True
      - Message
        - Seizure Detected: Minor (Sudden, repeated fear or anger)
    - Issue 3
      - Pattern
        - True, False, True, False, True
      - Message
        - Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)
    - Patient Stats
      - Signal 1: 100 to 130
      - Signal 2: 50 to 100
      - Signal 3: 80 to 150
      - Signal 4: 20 to 40
      - Signal 5: 130 to 180

3.5.2.2 Unit Testing

- Confirm that each class and its methods perform as expected
  - Test the modifying and returning of attributes
    - Test setters to modify attributes
      - Signal class must be able to set the value for the frequency
      - Issue class must be able to set values for the pattern and message
      - Patient class must be able to set values for the stats
        - isAbnormal is class exclusive so there is no setter
        - Stats must have a valid range. Tested using White-Box
    - Test getters to verify proper attribute values
      - Signal class must be able to get the value for the frequency
      - Issue class must be able to get values for the pattern and message

        - Patient class must be able to get values for the stats
            - isAbnormal is class exclusive so there is no getter
    - All the setters and getters behaved properly

3.5.2.3 White-Box Testing

Test all known paths of the code for valid and invalid inputs. We may assume that all values will work if the boundaries behave as expected. So, test every boundary for correctness.

Test for Patient stats range

- Equivalence Classes
    - Invalid Ranges
    - Valid Ranges
- Range must go from a smaller number to a larger number. (Ex: 80-100)
    - Test for when numbers are the same. (Valid)
        - Test Valid Signal 3 Stats: 103 to 103
            - Expected: Signal 3 Stats Set
            - Result: Signal 3 Stats Set
            - Conclusion: Test value behaves properly
    - Test for when smaller number is one greater than larger number (Invalid)
        - Test Invalid Signal 3 Stats: 103 to 2
            - Expected: "Signal 3 has an invalid range: 103 to 2"
            - Result: "Signal 3 has an invalid range: 103 to 2"
            - Conclusion: Test value behaves properly
    - Test for when smaller number is one less than larger number (Valid)
        - Test Valid Signal 3 Stats: 102 to 103
            - Expected: Signal 3 Stats Set
            - Result: Signal 3 Stats Set
            - Conclusion: Test value behaves properly

Test Entire Program with Default Test Values.

Test Patient Stat Range Boundaries

- Signal Abnormality Equivalence Classes
    - Signals below valid Patient Stat Ranges. (Abnormal)
    - Signals within valid Patient Stat Ranges. (Normal)
    - Signals above valid Patient Stat Ranges. (Abnormal)
- Test the boundaries of Patient Stat Ranges with the Signals.
    - Test All Signals for Lower Boundary (Within and Normal)
        - Signals are normal if they are within the boundaries.
            - LowerBoundary1(100) <= Signal1(100) <= UpperBoundary1(130) is True
                - Signal is normal so isAbnormal[0] = False
            - LowerBoundary2(50) <= Signal2(50) <= UpperBoundary2(100) is True
                - Signal is normal so isAbnormal[1] = False

        - LowerBoundary3(80) <= Signal3(80) <= UpperBoundary3(150) is True
            - Signal is normal so isAbnormal[2] = False
        - LowerBoundary4(20) <= Signal4(20) <= UpperBoundary4(40) is True
            - Signal is normal so isAbnormal[3] = False
        - LowerBoundary5(130) <= Signal5(130) <= UpperBoundary5(180) is True
            - Signal is normal so isAbnormal[4] = False
    - Results
        - Expected: isAbnormal is All False
        - Result: isAbnormal is All False
        - Conclusion: Test value behaves properly
- Test All Signals for Upper Boundary (Within and Normal)
    - Signals are normal if they are within the boundaries.
        - LowerBoundary1(100) <= Signal1(130) <= UpperBoundary1(130) is True
            - Signal is normal so isAbnormal[0] = False
        - LowerBoundary2(50) <= Signal2(100) <= UpperBoundary2(100) is True
            - Signal is normal so isAbnormal[1] = False
        - LowerBoundary3(80) <= Signal3(150) <= UpperBoundary3(150) is True
            - Signal is normal so isAbnormal[2] = False
        - LowerBoundary4(20) <= Signal4(40) <= UpperBoundary4(40) is True
            - Signal is normal so isAbnormal[3] = False
        - LowerBoundary5(130) <= Signal5(180) <= UpperBoundary5(180) is True
            - Signal is normal so isAbnormal[4] = False
    - Results
        - Expected: isAbnormal is All False
        - Result: isAbnormal is All False
        - Conclusion: Test value behaves properly
- Test All Signals for One Below the Lower Boundary (Below and Abnormal)
    - Signals are abnormal if they are not within the boundaries.
        - LowerBoundary1(100) <= Signal1(99) <= UpperBoundary1(130) is False
            - Signal is abnormal so isAbnormal[0] = True
        - LowerBoundary2(50) <= Signal2(49) <= UpperBoundary2(100) is False
            - Signal is abnormal so isAbnormal[1] = True
        - LowerBoundary3(80) <= Signal3(79) <= UpperBoundary3(150) is False
            - Signal is abnormal so isAbnormal[2] = True
        - LowerBoundary4(20) <= Signal4(19) <= UpperBoundary4(40) is False
            - Signal is abnormal so isAbnormal[3] = True
        - LowerBoundary5(130) <= Signal5(129) <= UpperBoundary5(180) is False
            - Signal is abnormal so isAbnormal[4] = True
    - Results
        - Expected: isAbnormal is All True
        - Result: isAbnormal is All True
        - Conclusion: Test value behaves properly.

- Test All Signals for One Above the Upper Boundary (Above and Abnormal)
  - Signals are are abnormal if they are not within the boundaries.
    - LowerBoundary1(100) <= Signal1(131) <= UpperBoundary1(130) is False
      - Signal is abnormal so isAbnormal[0] = True
    - LowerBoundary2(50) <= Signal2(101) <= UpperBoundary2(100) is False
      - Signal is abnormal so isAbnormal[1] = True
    - LowerBoundary3(80) <= Signal3(151) <= UpperBoundary3(150) is False
      - Signal is abnormal so isAbnormal[2] = True
    - LowerBoundary4(20) <= Signal4(41) <= UpperBoundary4(40) is False
      - Signal is abnormal so isAbnormal[3] = True
    - LowerBoundary5(130) <= Signal5(181) <= UpperBoundary5(180) is False
      - Signal is abnormal so isAbnormal[4] = True
  - Results
    - Expected: isAbnormal is All True
    - Result: isAbnormal is All True
    - Conclusion: Test value behaves properly

Test Issue Detection

- Test Issues Equivalence Classes
  - Issue 1 Detected (True, True, True, True, True)
  - Issue 2 Detected (True, False, False, False, True)
  - Issue 3 Detected (True, False, True, False, True)
  - Normal (Any pattern besides the test Issues)
- Test Issue Pattern Detection with the Abnormality Patterns
  - Issue 1
    - Issue 1 is caused when isAbnormal = {True, True, True, True, True}
      - LowerBoundary1(100) <= Signal1(90) <= UpperBoundary1(130) is False
        - Signal is abnormal so isAbnormal[0] = True
      - LowerBoundary2(50) <= Signal2(110) <= UpperBoundary2(100) is False
        - Signal is abnormal so isAbnormal[1] = True
      - LowerBoundary3(80) <= Signal3(40) <= UpperBoundary3(150) is False
        - Signal is abnormal so isAbnormal[2] = True
      - LowerBoundary4(20) <= Signal4(80) <= UpperBoundary4(40) is False
        - Signal is abnormal so isAbnormal[3] = True
      - LowerBoundary5(130) <= Signal5(10) <= UpperBoundary5(180) is False
        - Signal is abnormal so isAbnormal[4] = True
    - Results
      - Expected: "Seizure Detected: EMERGENCY (Complete bodily shutdown)"
      - Result: "Seizure Detected: EMERGENCY (Complete bodily shutdown)"
      - Conclusion: Test value behaves properly
  - Issue 2
    - Issue 2 is caused when isAbnormal = {True, False, False, False, True}
      - LowerBoundary1(100) <= Signal1(90) <= UpperBoundary1(130) is False

      - Signal is abnormal so isAbnormal[0] = True
    - LowerBoundary2(50) <= Signal2(75) <= UpperBoundary2(100) is True
      - Signal is normal so isAbnormal[1] = False
    - LowerBoundary3(80) <= Signal3(110) <= UpperBoundary3(150) is True
      - Signal is normal so isAbnormal[2] = False
    - LowerBoundary4(20) <= Signal4(30) <= UpperBoundary4(40) is True
      - Signal is normal so isAbnormal[3] = False
    - LowerBoundary5(130) <= Signal5(10) <= UpperBoundary5(180) is False
      - Signal is abnormal so isAbnormal[4] = True
  - Results
    - Expected: "Seizure Detected: Minor (Sudden, repeated fear or anger)"
    - Result: "Seizure Detected: Minor (Sudden, repeated fear or anger)"
    - Conclusion: Test value behaves properly
- Issue 3
  - Issue 3 is caused when isAbnormal = {True, False, True, False, True}
    - LowerBoundary1(100) <= Signal1(90) <= UpperBoundary1(130) is False
      - Signal is abnormal so isAbnormal[0] = True
    - LowerBoundary2(50) <= Signal2(75) <= UpperBoundary2(100) is True
      - Signal is normal so isAbnormal[1] = False
    - LowerBoundary3(80) <= Signal3(500) <= UpperBoundary3(150) is False
      - Signal is abnormal so isAbnormal[2] = True
    - LowerBoundary4(20) <= Signal4(30) <= UpperBoundary4(40) is True
      - Signal is normal so isAbnormal[3] = False
    - LowerBoundary5(130) <= Signal5(10) <= UpperBoundary5(180) is False
      - Signal is abnormal so isAbnormal[4] = True
  - Results
    - Expected: "Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)"
    - Result: "Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)"
    - Conclusion: Test value behaves properly
- Normal
  - Normal occurs when isAbnormal does not match any Issue Pattern (i.e. All False)
    - LowerBoundary1(100) <= Signal1(110) <= UpperBoundary1(130) is True
      - Signal is normal so isAbnormal[0] = False
    - LowerBoundary2(50) <= Signal2(75) <= UpperBoundary2(100) is True
      - Signal is normal so isAbnormal[1] = False
    - LowerBoundary3(80) <= Signal3(110) <= UpperBoundary3(150) is True
      - Signal is normal so isAbnormal[2] = False
    - LowerBoundary4(20) <= Signal4(30) <= UpperBoundary4(40) is True
      - Signal is normal so isAbnormal[3] = False
    - LowerBoundary5(130) <= Signal5(155) <= UpperBoundary5(180) is True
      - Signal is normal so isAbnormal[4] = False

- Results
  - Expected: "Normal"
  - Result: "Normal"
  - Conclusion: Test value behaves properly

White-Box Testing Conclusions

All test performed as expected. Every path the program was tested and performed each operation as expected.

Randomly Generated Signals Test

The Signals are determined using a random number generator between 0 and 200. These values are compared with the default Patient Stat Ranges. If the Signal's random number is out of its corresponding range, then it is determined to be abnormal. The abnormalities are compared with the three default Issue Patterns. The randomly generated signals represent what the program would be receiving once implemented. Thus, it is important to run and make sure the program functions accordingly. All random tests we run produced the expected output.

3.5.2.4 Testing Conclusions

1. All the Tests Passed
   a. The program operates as expected
2. Verification Confirmed
   a. All workflows were completed correctly
3. Validation Confirmed
   a. The product satisfies the requirements
4. Program is Ready for Integration
   a. Move to UML stage to determine compatibility with the previous team.

3.6 UML

3.6.1 Overview

Once all the workflows were completed, we created a UML diagram representing our program. The UML diagram displayed the three classes. The lines connecting the three classes together showed the relationship between them. Within the classes were the attributes used and the operations performed. The attributes displayed their types while the operations displayed their return and parameter types. Each class was unique and had its own purpose.

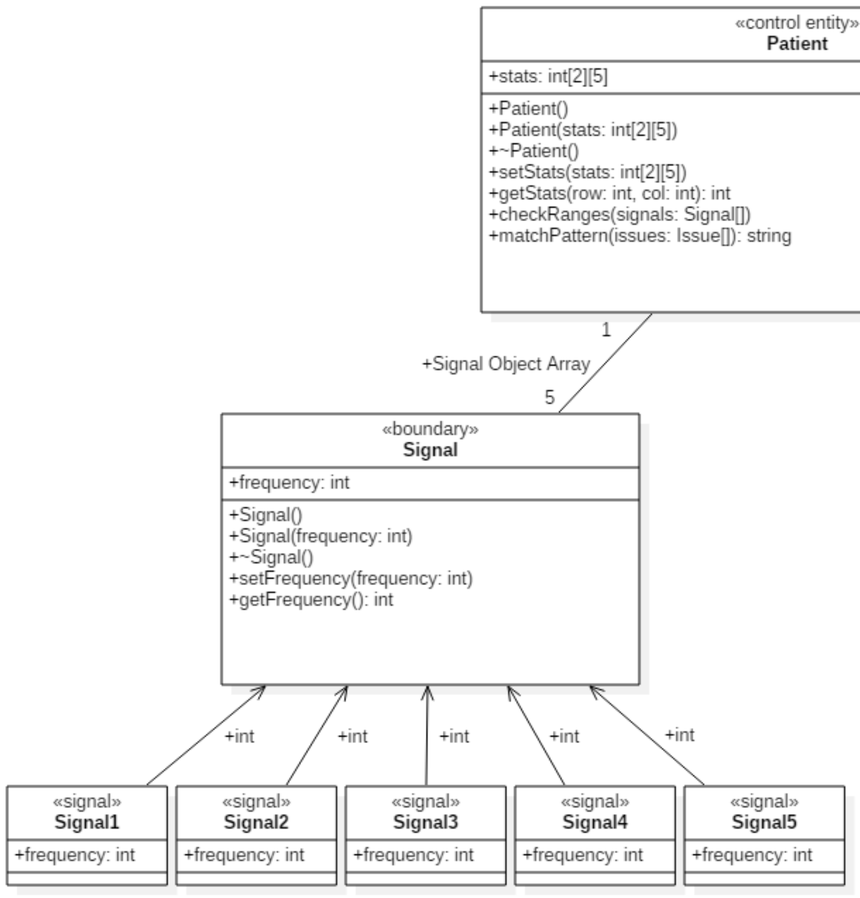
The Signal class interacted with five incoming signals which made it a Boundary class. These signals were each assigned to a Signal object. The frequency attribute within the Signal objects were constantly being updated by their associated signal using the setFrequency operation. A getFrequecy operation was also implemented so the Patient class would be able to access the attribute. The Signal objects were grouped together in a Signal object array of size five and used by the Patient class.

The Issue class created objects used to hold information regarding an Issue. Message and pattern were the two attributes in the Issue class that represented an Issue's information. The message embodied what was causing a seizure and the pattern was used as a Boolean array of abnormalities that would denote the Issue. Each attribute had its own setter operation, so it could be modified. The Issue attributes also had corresponding getter operations, so the Patient class could access an Issue object's information. The Issue objects were grouped together in an Issue object array of any size and used by the Patient class.

The Patient class was where the Signal and Issue objects were used. The Signal and Issue classes did not communicate directly with each other. Instead, they were controlled by the Patient class. The Patient class held the stats 2D Integer Array which was compared with the incoming Signal objects. The stats and five Signal objects were compared using the checkRanges operation. The checkRanges operation detected whether a signal was abnormal or not and inputted it into an isAbnormal Boolean array. Then, the isAbnormal Boolean array was compared with any amount of Issue objects to determine if and Issue was occurring using the matchPattern operation. The matchPattern operation outputted whether the Signal objects were normal or represented an Issue object. Thus, the Patient class operated as both an Entity class and Control class.

The purpose of creating the UML was to determine if our program was compatible with the previous teams. However, when we conferred with the previous team, we discovered that our UML was not compatible with theirs. They were the ones determining if the signals were abnormal or not. Therefore, we no longer needed to check Signals with Patient stats. They told us that they would send five Boolean signals, so we could associate them with issues. Subsequently, we modified our project to accommodate this change. Section 4 will step through all the changes we made to the workflows.

3.6.2 UML Diagram



4 FINAL PROJECT

4.1 REQUIREMENTS

Requirements Supervisor: Clayton D. Terrill

4.1.1 Overview

When we discovered we had conflicts with the previous team, we had to rebuild our requirements. We listened to exactly what the previous team was doing. Then, we talked with them about possibly resolving the conflicts. Our team and the previous team came to an agreement that the previous team will send us five Boolean signals. The Boolean signals will represent if they detected an abnormality or not. So, all we will have to do is to match these Boolean signals with Issue patterns. Using this information, we drafted some new requirements which cut down on what our final portion of the project will accomplish.

Our new requirements outline that we will receive five signals representing Boolean values. These signals will then be directly compared with patterns that represent a specific issue. Each issue is associated with an epileptic seizure. So, if an issue pattern matches the incoming signals, then we output the issue. These new program requirements make our program much simpler. All we must do is modify the other workflows to reflect the changes in the requirements.

4.1.2 Contract

There are five signals being received and monitored. Each signal carries whether an abnormality was detected or not. The signals are used together and matched to known issue patterns that cause epileptic seizures. If the signals are associated with an issue pattern, then an epileptic seizure is predicted, and a message is produced.

Group Member Requirements:

- All group members are required to attend every workshop class and is responsible for their assigned job.
- The project will be written in C++.
- The project will be written from scratch by the group members.
- Group members will write a report regarding the project.
- Group members will create a PowerPoint to present.
- The team leader will make sure that all the workflows are completed accordingly.
- The team leader will compile the final report and project.
- If the requirements change, then each group member is required to accommodate them.
- All workflows will be completed prior to Tuesday, April 17th, 2018.
- A presentation will be presented by the group leader von Thursday, April 19th, 2018.
- A full report will be turned in prior to Monday, April 20th, 2018.

Names	**Jobs**	**Status**
Clayton Terrill	Requirements and Group Leader	**Active**
Elliot Fries	Analysis	**Active**
James Languirand	Design	**Active**
Ian Barney	Implementation	**Active**
Jeremy Watson	Testing	**Missing**

4.2 ANALYSIS

Analysis Supervisor: Elliot C. Fries

4.2.1 Overview

The final analysis is much simple. In our original analysis, we had eight nouns that could be classes or attributes. However, in our final analysis, we only have six nouns which may be classes or attributes. These nouns include Signal, Abnormality, Issue, Pattern, Seizure, and Message. The project is stripped of the need to keep track of patient statistics and determine if the incoming signals are abnormal or not. Now, the requirements only need to compare the incoming signals with the issue patterns. Consequently, there is less to describe and fewer nouns are needed. Based on the new set of nouns, we kept two classes the same while replacing one of them. The new three classes include Signal, Issue, and Seizure.

The Signal class handles the incoming signals and stores whether an abnormality was previously detected. The Signal class will still have multiple instances and they will continuously read from incoming signals. Therefore, the purpose of the Signal class remains the same and acts as a boundary class that receives input from an outside source.

The Issue class does not change at all. The Issue class still has a pattern and a message. The pattern represents what signals must be abnormal for the issue to occur. If an issue occurs, then a meaningful message describing the issue must accommodate it. Therefore, the Issue class is still exclusively an entity class which contains a pattern and a message.

The Seizure class replaces the Patient class. The purpose of the Seizure class is to compare the signals with the issue patterns. If the signals and issue patterns match, then the issue message is outputted. Therefore, the seizure class acts exclusively as a control class by performing the comparison operation. This improves the cohesion and only data is being passed between the classes.

4.2.2 Noun Class Extraction

Abnormalities within a **patient's life signals** may warn of an upcoming epileptic **seizure**. So, there will be five **signals** that the **frequency** will be monitored for. The **signal frequencies** will be compared to the **patient's** normal **statistics** to determine if there is an **abnormality**. **Abnormalities** within the **signals** may match a **pattern** that is associated with an **issue**. If an **issue** is detected, a **message** for the epileptic **seizure** will be outputted.

- Nouns:
    - **Abnormalities, Signal**, **Seizure, Frequency, Statistics, Pattern, Patient**, **Issue**
        - **Abnormalities** is an abstract noun which is associated with a patient. Abnormalities are associated with patterns within the Patient class. Therefore, use abnormalities as an attribute within the Patient class.
        - **Frequency** is an abstract noun which is associated with a signal. Frequency represents the integer value being received in the Signal class. Therefore, use frequency as an attribute within the Signal class.
        - **Statistics** is an abstract noun which is associated with a patient. Statistics represents the normal frequency values for a patient's vital signals. Therefore, use statistics as an attribute within the Patient class.
        - **Seizure** is an abstract noun which is associated with issues. Issues describes what is causing the seizure. Therefore, a seizure class or attribute is not needed.
        - **Pattern** is an abstract noun which is associated with an issue. Each issue has a pattern of abnormalities associated with it. Therefore, use pattern as an attribute for the issue class.
        - **Message** is an abstract noun which is associated with an issue. Each issue needs to be identified. Therefore, use message as an attribute for the issue class.
- Patient Class
    - Description
        - Creates object representing the patient.
    - Entity Class and Control Class
        - Holds the normal stats for the patient and compares them with the signal.
    - Attributes
        - Abnormalities – Represents if an abnormality was detected within a signal.
        - Statistics – Normal signal frequencies for the patient.
- Signal Class
    - Description
        - Creates objects representing the Integer values being received from the signals.
    - Boundary Class
        - Receives input from an outside source.
    - Attributes
        - Frequency – The frequency being received from the signal.
- Issue Class
    - Description
        - Creates objects representing the issue patterns.
    - Entity Class
        - Holds the pattern of abnormalities representing the issue.
    - Attributes
        - Pattern – Represents what pattern of signals causes the issue.
        - Issue Message – The output describing what issue is occurring.

4.3 DESIGN

Design Supervisor: James S. Languirand

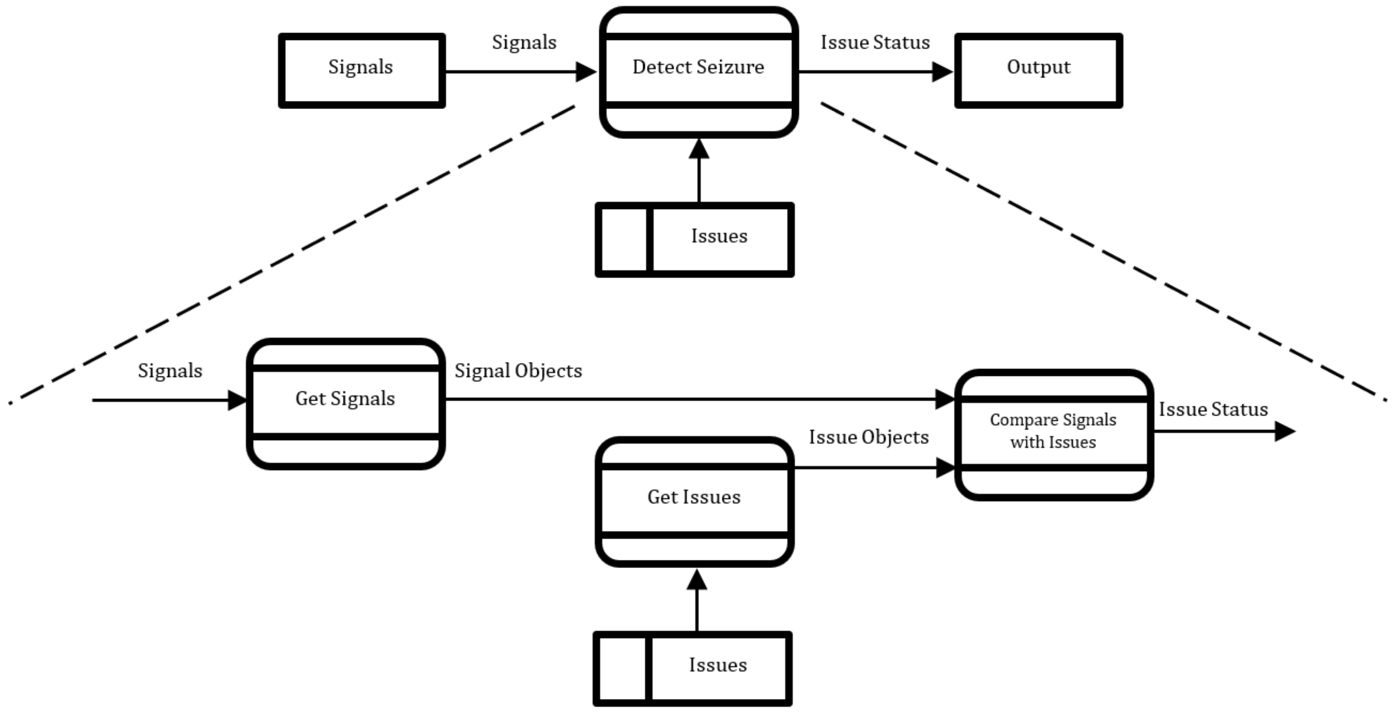
4.3.1 Overview

The Design workflow needs to be modified to reflect the disappearance of processes. We still used a top-down decomposition chart to model the flow of data. The only part of the decomposition being removed is where we detect abnormalities. The removal of detecting abnormalities results in only two levels of decomposition. The new design models the flow of data and provides a blueprint for an implementer so they modify the program.

The top level of decomposition consists of three major parts. The first part is the incoming signals. This is the outside source we will be receiving. The signals will enter our detect seizure process. Here is where we perform operations to see if an Issue is occurring. Then, we will output the Issue status by returning the Issue message. The top level is an overview of what the program does and may be decomposed even farther.

The second level of decomposition outlines what needs to be created and loaded to detect an issue. This level converts the incoming signals to objects while also loading the issue objects. The Signal objects now hold the isAbnormal Boolean values representing the signal. The Signal objects are grouped into an array and passed to the Seizure class. Meanwhile, the Issue objects are created on startup using input from a file or user. These objects contain a pattern of abnormalities and a message representing the issue. The Issue objects are also grouped into an array and compared with the Signal objects inside the Seizure class. So, the Seizure class only performs one operation which means it does not need to be decomposed farther.

4.3.2 Top-Down Decomposition Data Flow Diagram



4.4 IMPLEMENTATION

Implementation Supervisor: Ian S. Barney

4.4.1 Overview

With the simplification of the previous workflows, the implementation is reduced. The implementation workflow now consists of performing modifications to the original program. We decided that it would be best to track the modifications made. Therefore, we use a source control called GitHub. Additionally, this allows us to use parallel development and efficiently make the changes.

The first step to modify the code is to make the classes reflect the new outline in the analysis. So, the Patient class is now removed, and the Seizure class is added. Attributes are modified to match the types of variables needed in each class. The methods are also modified to replicate the variable types. There are slight modifications in the Signal class, none in the Issue class, and the whole Seizure class replaces the Patient class. Additionally, improvements are made to allow pointers.

The Signal is still a Boundary class for the program. So, it still has the necessary setters and getters for the variables. The only difference is the variable type. We now have a Boolean value referred to as isAbnormal instead of an integer value called frequency. IsAbnormal represents if the incoming signal represents an abnormality or not. Since there are still five signals being received, this Signal class produces five objects.

The Issue class has not changed at all. It is still an Entity class because it holds information regarding a specific issue. The variables being used are message and pattern. The message variable represents what the name of the issue is. The purpose behind the message is to display what is causing and epileptic seizure. The pattern variable represents the pattern of abnormalities that are associated with the issue. There is still no limit on how many Issue objects that may be created. If two objects share a pattern, then our implementation will only display the first one which was created. If an Issue needs to display a different message or use a different pattern, then the setters provided may be used to modify the variables. The getters continue to allow access to the variables from the control portion of the Seizure class.

The Seizure class is the new class in the program replacing the Patient class. There is no longer any need to store information within the class, so it is exclusively a Control class. All the Seizure class does is compare the incoming signals with all the Issue objects. If the signals indicate an issue, then the issue’s message is returned. The issue message indicates the Issue that is causing an epileptic seizure.

4.4.2 Code Documentation

4.4.2.1 Signal.h

- Overview
  - Declares the variables and function signatures used in the Signal.cpp.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Variables
  - isAbnormal
    - Boolean value that shows if signal is abnormal or not.
- Function Signatures
  - Signal();
  - Signal(bool);
  - ~Signal();
  - void setIsAbnormal(bool);
  - bool getIsAbnormal();

4.4.2.2 Signal.cpp

- Overview
  - Signal Class is used to create a Signal object that monitors signals being recieved. The Signal object's isAbnormal Boolean value will be constantly changed to represent if the Signals are producing an abnormality or not.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Methods:
  - Signal()
    - Default Constructor of the Signal. Initializes isAbnormal to be false.
  - Signal(bool isAbnormalTemp)
    - Constructor for when a value has been designated during Signal object creation. Sets the isAbnormal.
    - @param isAbnormalTemp - Boolean value to set isAbnormal with.
  - ~Signal()
    - Default Destructor that deletes the Signal. Prevents Memory Leak.
  - setIsAbnormal(bool isAbnormalTemp)
    - Sets the isAbnormal variable with a boolean value.
    - @param isAbnormalTemp - Boolean value to set isAbnormal with.
  - getIsAbnormal ()
    - Returns whether a signal is abnormal or not.
    - @return isAbnormal - Boolean value that shows if signal is abnormal or not.

4.4.2.3 Issue.h

- Overview
    - Declares the variables and function signatures used in the Issue.cpp.
- Authors
    - Clayton D. Terrill and Ian S. Barney
- Variables
    - message
        - String value that displays the Issue.
    - pattern
        - Boolean array that represents the Issue Pattern.
    - Function Signatures
    - Issue();
    - Issue(bool*, string);
    - ~Issue();
    - void setPattern (bool*);
    - void setMessage(string);
    - bool* getPattern();
    - string getMessage();

4.4.2.4 Issue.cpp

- Overview:
    - Issue Class is used to create an Issue object that represents an Issue Pattern. The Issue Pattern will be compared with to see if that specific Issue is occurring.
- Authors:
    - Clayton D. Terrill and Ian S. Barney
- Methods:
    - Issue()
        - Default Constructor of the Issue. Initialize the size of the boolean array to 5. Initializes the pattern to be all true. Initializes the message as 'Detected an Issue'.
    - Issue(bool* patternTemp, string messageTemp)
        - Constructor for when a boolean array and message has been designated during the Issue object creation. Sets the pattern array. Sets the message.
        - @param patternTemp - The array to set the pattern array with.
        - @param messageTemp - String value that contains the Issue.
    - ~ Issue()
        - Default Destructor that deletes the Issue. Prevents Memory Leak.
    - setPattern(bool* patternTemp)
        - Sets the pattern array.
        - @param patternTemp - The array to set the pattern array with.
    - setMessage(string messageTemp)
        - Sets the message variable with a String value.
        - @param messageTemp - String value that contains the Issue.
    - getPattern()
        - Returns Issue Pattern.
        - @return bool* - Pointer to the Boolean pattern array.

- getMessage()
  - Returns the Issue message.
  - @return string - String containing the Issue message.

4.4.2.5 Seizure.h

- Overview
  - Declares the variables and function signatures used in the Seizure.cpp.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Variables
  - NONE
- Function Signatures
  - Seizure();
  - ~Seizure();
  - String checkForSeizure(Signal*, Issue*);

4.4.2.6 Seizure.cpp

- Overview
  - Seizure Class is used as a control class that compares the Signal objects with the Issue object patterns and determines if there is a problem being outlined by the signals.
- Authors
  - Clayton D. Terrill and Ian S. Barney
- Methods:
  - Seizure()
    - Default Constructor of the Issue. Only needed to access the class.
  - ~ Seizure()
    - Default Destructor that deletes the Seizure class. Prevents Memory Leak. No memory allocated in this class, so no deallocation is needed.
  - checkForSeizure(Signal* signalArray, Issue* issueArray)
    - Compares the Signals in the Signal array with each Issue's pattern.
    - @param signalArray - Pointer to the signalArray which allows access.
    - @param issueArray - Pointer to the issueArray which allows access.
    - @return bool* - Pointer to the Boolean pattern array.

4.4.3 Online Code Link

- The code and project artifacts were uploaded to GitHub for easy access.
  - https://github.com/Terrillc13/PatternsMatchingAssociationPrediction/tree/FinalProject
- The code and project artifacts will also be included in the .zip file.

4.5 TESTING

Testing Supervisor: Clayton D. Terrill

4.5.1 Overview

The Testing workflow is now easier because the class that requires the most testing has been removed. We still use Testing to verify that every workflow is properly completed and produces a valid product. The product must fulfill the new requirements. So, functionality for each class and the overall program is to still be tested. The White-Box Testing is greatly reduced. Thus, the test harness is modified and reflects the changes implemented.

The test harness carries out all the new tests. We still need to make sure each class works as expected. Each class will have to be tested again to assure that the values in them still behave as projected. So, we will use Unit Testing again. The class variables will all be tested with setters and getters.

The Signal class needs to be able to constantly update its frequency value using a setter. Then, the Seizure control class needs to read the Signal's isAbnormal. So, the getter must be tested to assure it returns the appropriate value. The Issue class needs to be able to hold a pattern and message. Thus, the functions to set the Issue variables must be checked. Then, the Seizure control class needs to read the pattern and message from the Issue class. So, the getter for both variables must also be tested. The Seizure control class does not have any variables. The Seizure class is data dependent on the other two classes. All that needs to be tested in the Patient class is that it detects the appropriate Issue.

The White-Box Testing performed on the Seizure class is not nearly as complex as the Patient class was. There are only Boolean value comparisons now. Boolean values may be true or false. So, those are the only two cases we need to test for. All we must do now is to modify what the signals produce and match them with test issues. Using these tests, we can prove that our program works as expected.

4.5.2 Tests

4.5.2.1 Test Harness

- Created a test harness in the main to perform every test
    - Default Test Values Used for Various Tests
        - Issue 1
            - Pattern
                - True, True, True, True, True
            - Message
                - Seizure Detected: EMERGENCY (Complete bodily shutdown)
        - Issue 2
            - Pattern
                - True, False, False, False, True
            - Message
                - Seizure Detected: Minor (Sudden, repeated fear or anger)
        - Issue 3
            - Pattern
                - True, False, True, False, True
            - Message
                - Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)

4.5.2.2 Unit Testing

- Confirm that each class and its methods perform as expected
    - Test the modifying and returning of attributes
        - Test setters to modify attributes
            - Signal class must be able to set the value for the isAbnormal Boolean value
            - Issue class must be able to set values for the pattern and message
            - Seizure class does not have any setters.
        - Test getters to verify proper attribute values
            - Signal class must be able to get the value for the isAbnormal Boolean value
            - Issue class must be able to get values for the pattern and message
            - Patient does not have any getters.
    - All the setters and getters behaved properly

4.5.2.3 White-Box Testing

Test all known paths of the cod. We may assume that all values will work if the boundaries behave as expected. So, test every boundary for correctness.

Test Entire Program with Default Test Values.

Test Issue Detection

- Test Issues Equivalence Classes
    - Issue 1 Detected (True, True, True, True, True)
    - Issue 2 Detected (True, False, False, False, True)
    - Issue 3 Detected (True, False, True, False, True)
    - Normal (Any pattern besides the test Issues)
- Test Issue Pattern Detection with the Abnormality Patterns
    - Issue 1
        - Issue 1 is caused when isAbnormal = {True, True, True, True, True}
            - Signal1(True)
                - Signal is abnormal so isAbnormal[0] = True
            - Signal2(True)
                - Signal is abnormal so isAbnormal[1] = True
            - Signal3(True)
                - Signal is abnormal so isAbnormal[2] = True
            - Signal4(True)
                - Signal is abnormal so isAbnormal[3] = True
            - Signal5(True)
                - Signal is abnormal so isAbnormal[4] = True
        - Results
            - Expected: "Seizure Detected: EMERGENCY (Complete bodily shutdown)"
            - Result: "Seizure Detected: EMERGENCY (Complete bodily shutdown)"
            - Conclusion: Test value behaves properly
    - Issue 2
        - Issue 2 is caused when isAbnormal = {True, False, False, False, True}
            - Signal1(True)
                - Signal is abnormal so isAbnormal[0] = True
            - Signal2(False)
                - Signal is normal so isAbnormal[1] = False
            - Signal3(False)
                - Signal is normal so isAbnormal[2] = False
            - Signal4(False)
                - Signal is normal so isAbnormal[3] = False
            - Signal5(True)
                - Signal is abnormal so isAbnormal[4] = True
        - Results
            - Expected: "Seizure Detected: Minor (Sudden, repeated fear or anger)"
            - Result: "Seizure Detected: Minor (Sudden, repeated fear or anger)"
            - Conclusion: Test value behaves properly
    - Issue 3
        - Issue 3 is caused when isAbnormal = {True, False, True, False, True}

        - Signal1(True)
            - Signal is abnormal so isAbnormal[0] = True
        - Signal2(False)
            - Signal is normal so isAbnormal[1] = False
        - Signal3(True)
            - Signal is abnormal so isAbnormal[2] = True
        - Signal4(False)
            - Signal is normal so isAbnormal[3] = False
        - Signal5(True)
            - Signal is abnormal so isAbnormal[4] = True
    - Results
        - Expected: "Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)"
        - Result: "Seizure Detected: Minor (Repeated, unusual movements such as head nodding or rapid blinking)"
        - Conclusion: Test value behaves properly
- Normal
    - Normal occurs when isAbnormal does not match any Issue Pattern (i.e. All False)
        - Signal1(False)
            - Signal is normal so isAbnormal[0] = False
        - Signal2(False)
            - Signal is normal so isAbnormal[1] = False
        - Signal3(False)
            - Signal is normal so isAbnormal[2] = False
        - Signal4(False)
            - Signal is normal so isAbnormal[3] = False
        - Signal5(False)
            - Signal is normal so isAbnormal[4] = False
    - Results
        - Expected: "Normal"
        - Result: "Normal"
        - Conclusion: Test value behaves properly

White-Box Testing Conclusions

All test performed as expected. Every path the program was tested and performed each operation as expected.

Randomly Generated Signals Test

The Signals are determined using a random Boolean generator that sets each signal as true or false. These values are compared with the three default Issue Patterns. The randomly generated signals represent what the program would be receiving once implemented. Thus, it is important to run and make sure the program functions accordingly. All random tests we run produced the expected output.

4.5.2.4 Testing Conclusions

5. All the Tests Passed
    a. The program operates as expected
6. Verification Confirmed
    a. All workflows were completed correctly
7. Validation Confirmed
    a. The product satisfies the requirements
8. Program is Ready for Integration
    a. Move to UML stage to determine compatibility with the previous team.

4.6 UML

4.6.1 Overview

Once all the workflows were modified, we created a UML diagram representing our new program. The UML diagram displays the updated three classes. The lines connecting the three classes together show the relationship between them. Within the classes are the final attributes used and the operations performed. The attributes display their types while the operations display their return and parameter types. Each class is still unique and has its own purpose.

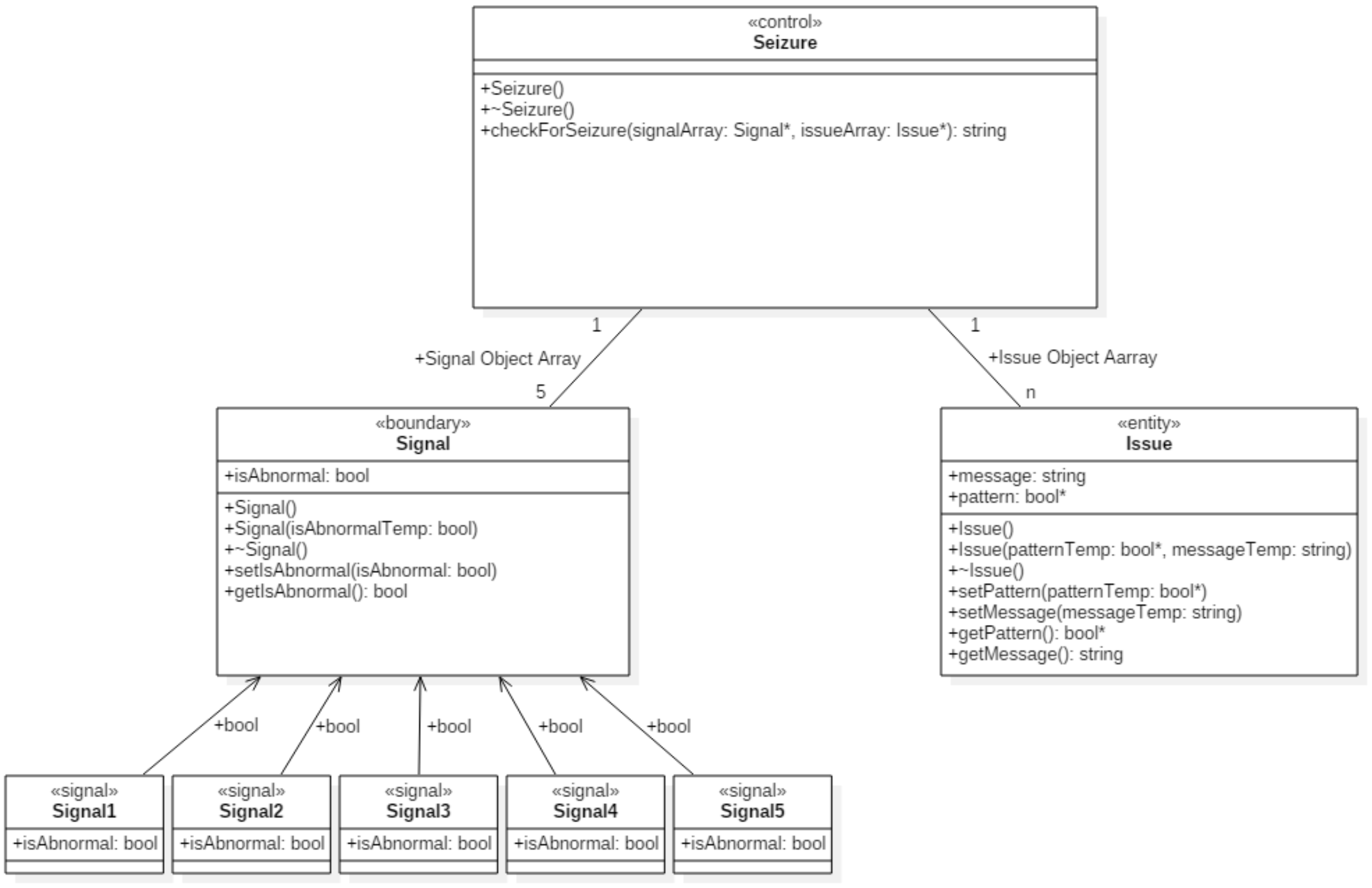
The Signal class interacts with five incoming signals which makes it a Boundary class. These signals are each assigned to a Signal object. The only aspect that changed from the original UML was the attributes. The frequency attribute within the Signal objects no longer exists. Instead, the isAbnormal Boolean values will be constantly updated by their associated signal using the setIsAbnormal operation. A getIsAbnormal operation will now provide the Seizure class a way to access the Boolean value. The Signal objects are still being grouped together in a Signal object array of size five. The Signal array will be used by the Seizure class.

The Issue class has not changed. It still creates objects to hold information regarding an Issue. Message and pattern are the two attributes in the Issue class that represent an Issue's information. The message embodies what was causing a seizure and the pattern is used as a Boolean array of abnormalities that denote the Issue. Each attribute has its own setter operation, so it can be modified. The Issue attributes also have corresponding getter operations, so the Seizure class may access the Issue object's information. The Issue objects are still being grouped together in an Issue object array of any size. The Issue array will also be used by the Seizure class.

The Seizure class is where the comparisons between the Signal and Issue objects are occurring. The Signal and Issue classes do not communicate directly together. Instead, the Seizure class compares the five Signal objects with any amount of Issue objects to determine if and Issue is occurring. The Seizure class does this by using the checkForSeizure operation. The checkForSeizure operation will output whether the Signal objects are normal or represent an Issue object. Thus, the Seizure class operates as the Control class.

The purpose of redoing the UML is to make our program compatible with the previous teams. Our original UML was not compatible. So, the new UML is created to fix the incompatibility. Using the new UML, our requirements change. Since the requirements change, our other workflows must be modified. Therefore, the final workflows are modifications of the original workflows. The modifications made result in our final program being able to successfully integrate with the previous team's program.

4.6.2 UML Diagram



5 CONCLUSION

We were tasked with the Patterns Matching, Association & Prediction of the Wearable Epilepsy Device & MC Team. We went through the complex process of completing just a portion of an entire project. Even though we used the Unified Process, we still faced many issues. One issue is that we lost a group member which meant the other members had to do more work. Furthermore, we wanted to make sure our finished program was able to be integrated with the team before us. So, we ended up with two versions of our program and workflows. One with the initial requirements, and the other with the revised requirements.

The initial requirements had us detecting abnormalities within signals using a Patient's normal signal ranges. However, when we got to the UML stage we were informed that the previous team was performing this operation. So, now we had a moving target problem. Our requirements changed, and we had to modify every workflow. This proved to be a grueling task, but we wanted to successfully complete this project. So, we made the modifications and produced a final program suitable for integration.

The only step left is to integrate our program with the other teams within the Wearable Epilepsy Device project. To integrate our program, two methods would need to be created. The first method would need to be called on startup. This method would either load the Issues from a file or have a user input them. The second method would need to create the Signal objects and constantly update them with the incoming signals. This second method would also need to call the checkForSeizure method within the Seizure control class. Correspondingly, the Signal objects and Issue objects would need to be passed into the checkForSeizure method each time it was called. Looking back on this, we could have implemented a listener to automatically run each time a Signal object was modified. However, we did successfully implement the classes and flow of data. To ensure that our portion of the project behaved properly, we thoroughly tested it using various testing methods. Using the Unified Process, we successfully fashioned our portion of the project. Yet, we ran into one of the biggest issues that could have occurred during the UML stage. Our portion of the project was not compatible with the team before us.

Since we are the final team within the Wearable Epilepsy Device project, our job was to receive five signals from the previous team and predict if an epileptic seizure was occurring. We completed our task proficiently and correctly. We learned a lot about how teams within a project operate and the struggles when an organization does not communicate well with others. Due to the lack of communication, we had to extensively modify our program so it could be integrated with the other team's program. We learned firsthand how frustrating changes in the requirements can be. However, we persisted, and we were able to make the final program fulfill the final requirements completely.

6 REFERENCES

Schach, S. R. (2011). *Object-Oriented and Classical Software Engineering* (Eighth ed.). New York, NY: McGraw-Hill.