Internet of Things Solution

For Hugs the Rail Transportation Co.

Christopher Bowden
Yegor Kozhevnikov
Nolan Vernon

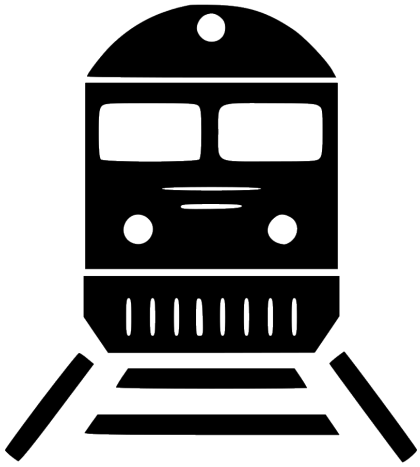


Table of Contents

1. Communication and Planning

1.1. Problem Statement:

To create a safe and efficient system using an Internet of Things approach to append the current Locomotive Control System in the absence of an Internet connection.

1.2. Stakeholders:

Hugs the Rail Transportation Company is the biggest stakeholder for this project as the trains that this system will be deployed on are running under their name. The solution must not endanger the property or the people using said trains. The other stakeholder party is of the people who rely on the trains to deliver goods, services, and other people. Their interests and opinions will have to be factored in as well, because their financial stability may rely on Hugs the Rail's stability.

1.3. Audience:

The biggest portion of the targeted audience for this product will be the train operators which will have to interact with it. It is important for us to make the IoT solution as easy to understand and as streamlined as possible for this reason. Another major part of the audience will be the technicians. They will have to be able to update the system, diagnose and fix problems in it, and do general maintenance.

1.4. Importance and value to users:

To keep the railway system running smoother, meaning more guarantee that trains will arrive on time with less chance that unforeseen circumstances on the rails will result in accidents. Help maintain locomotive engines by running active diagnostics along with the train's regular functions and inform the operator of any deviations from the normal. Allow for operators to have more information, by synthesizing the sensor data by the processing unit, the operator will be able to have more knowledge about their running train and will be able to receive suggestions from it based on currently available information.

1.5. Approach to solution:

Our general approach will be to analyze what is required from the system and base solution to handle specific problems. Once that is made to a sufficient degree, we may advance to other, side upgrades to the system.

1.6. Timeline:

Communication and Planning	2/16
Requirement Analysis	3/15
Requirement Modeling	3/28
Software Architecture Designing	4/11

Coding	4/30
Testing	5/04
Deployment	5/03

1.7. Process Model: Prototyping Process Model

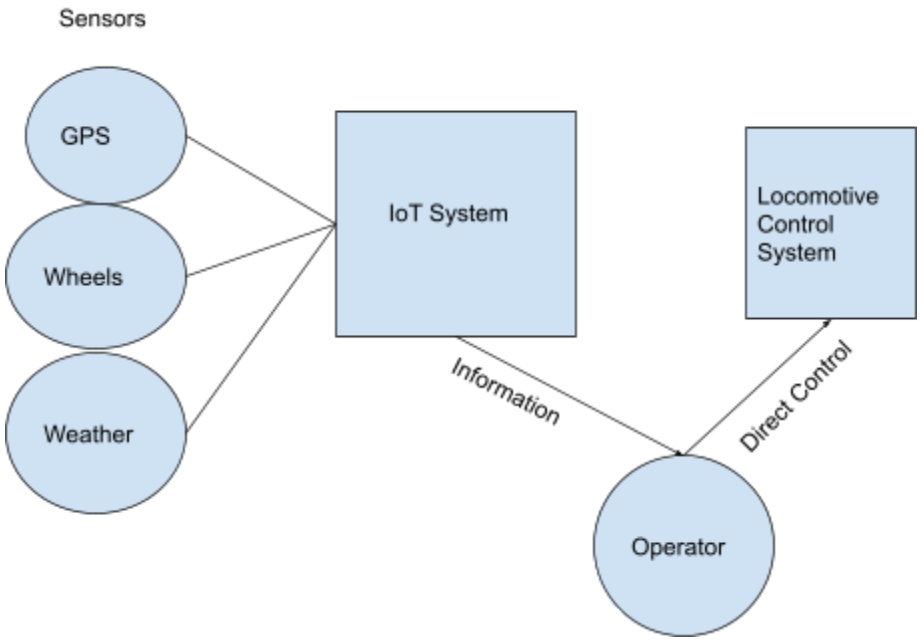
- Allows us to constantly evolve our plan and test elements of the solution.
- Very open-ended project - as requirements and our understanding of them change, we can adapt more quickly and efficiently
- Waterfall Model works better for larger organizations, testing late in the process could be a risk with a hard deadline
- Unified Model and Spiral Model require extensive customer involvement, which we don't have

2. Overview of the Control Center

2.1. Summary of the Problem:

The modern Locomotive Control System uses the Internet in order to maintain itself and have access to the broader array of information that it may require over the course of its use. At some points in the route, the Locomotive Control System may experience a lack of such connection and will no longer be able to rely on the network for information. This can prevent the system from knowing where there are other locomotives on the railway system, under what conditions is the current system running under, or how the information that it already has access to should be processed. By proposing an Internet of Things solution, in the absence of an outside connection to the Internet, the train will be able to collect its own information and depend on it, while it waits for the connection to be re-established. The IoT System will run along with the LCS and append its decision making with its own data that it will be able to gather. In the offline mode, the operator will have control over the Control Center and the LCS due to the nature of the information being inherently less reliable than the culmination of what the data center can supply. The IoT solution will have to be reliable enough to base concrete decisions on.

2.2. Fortunately there are technologies that can help us, a multitude of detection sensors can be installed onto the train in order to detect foreseeable events and have the engine react to them before they occur.



2.3. The proposed IoT engine solution will be able to take in input from installed sensors onboard the locomotive, pass data to the central processing unit, pass information to the operator about its actions, and take input from the operator for possible overrides and custom commands.

2.4. What IoT can provide:

- Proximity sensors around the train can detect the distance between stationary and moving objects on the railway; and set the rules for alerting the operator.
- Verify that railroad crossing gates are closed for crossing with sensors looking for gate lights and alarm bells.
- Collect data from wheel RPM and GPS position to detect whether or not the wheels are slipping; set rules on pouring sand on rails to increase grip
- Take temperature and humidity readings from the surrounding environment to see if ice can be formed on the railroad tracks; set under which conditions should the operator be alerted to this.
- In sum, have multiple layers of checks to ensure safety even without cellular or WiFi communication

3. Requirements

3.1. Non-Functional Requirements

3.1.1. Reliability

- IoT HTG should be operable under extreme weather conditions in temperatures ranging from -50 to 150 F°.
- IoT HTG shall stand drops up to 5 feet.
- IoT HTG Sensor power supply must withstand typical forces that is experienced by a moving locomotive.
- IoT HTG system shall have reliability of 0.999.
- IoT will self report any internal errors and log them to the log file

3.1.2. Performance

- IoT shall have a response time of 0.5 second while the device is operational.
- IoT shall be able to support up to 1000 sensors.
- Must be able to multi-thread incoming data from multiple sensors
- IoT's response time shall not increase when under load.
- The operator must be able to log into the IoT within 30 seconds of it turning on.

3.1.3. Security

- IoT shall be accessed only by User ID/Password, and/or Fingerprint ID
- Users will be encouraged to change passwords on a monthly basis to ensure security.
- IoT shall not be able to be accessed from an outside source.
- IoT software will be updated when needed, at least once per 4 months, with a hardwired connection to a device with an updated version.
- Communication between sensors and IoT device shall be secure and verifiable.
- All data going into the IoT device from sensors and all conclusions the device will be making will be logged.

3.1.4. Operating System

- The IoT will run on a portable operating system, due to the possibility of it being moved from train to train
- The IoT will run on a low-maintenance operating system, due to its operators not being trained on maintaining it
- The IoT will run on an efficient operating system, due to all processes being relevant to the interpretation of sensor data.

3.2. Functional Requirements

3.2.1. Detect standing objects on the path with distance and suggest speed changes or brake

- Will have doppler radar sensors installed at the front and back of the train.

- The IoT system will connect to the sensors when a conductor logs in.
- Only when the train is moving faster than 5 mph, will the IoT read from the sensors
- Sensor will periodically send out pulses at a certain frequency and wait for them to return.
- When a pulse of similar frequency is detected, sensors will report this to the IoT.
- IoT will calculate distance to object based on time it took for the pulse to come back and its speed known by the sensor
- If the distance is less than 10 miles, an inform message will be sent
- If the distance is less than 5 miles, an alert message will be sent.

3.2.2. Detect moving objects and its speed on the path, ahead or behind, and suggest speed changes of brake

- Similar to how to detect standing objects.
- IoT will require a GPS sensor to be installed.
- IoT will periodically check the difference in coordinates supplied by the sensor and calculate the train's speed.
- When the doppler sensor senses a returning pulse, if the frequency of the detected pulse is shorter than the sent pulse, the IoT will calculate the speed difference between the train and the detected object.
- IoT will request train's speed from the GPS and add it to the speed difference found previously.
- If the object in front is slower than the train, send a signal based on its proximity.
- If the object behind the train is faster than the train, send a signal based on its proximity

3.2.3. Detect gate crossing open/closed and distance, and inform the operator

- Will require a camera and parabolic microphone combinations sensor that can calculate distance to sound source.
- IoT will periodically poll the sensors for the detection of gate crossing lights and/or alarm bells.
- As long as the sensors detect the gate, the IoT will be aware of it and the distance to it.

- When a crossing gate is detected, IoT will inform the conductor and suggest to visually confirm it.
- Depending on the distance to the gate, IoT will display a message to blow the horn for a varied amount of time.
- When the gate is detected to be less than 1.5 miles away, IoT will display a message for 15 seconds to blow the horn.
- When the gate is detected to be less than 0.3 miles away, IoT will display a message for 5 seconds to blow the horn.

3.2.4. Detect wheel slippage and the amount of slippage, and inform the operator

- Will require GPS sensor and wheel RPM sensors
- IoT will periodically poll the GPS for location and calculate train speed based on the difference in coordinates.
- IoT will periodically poll RPM sensors, and take the most common RPM value as a basis for the average RPM of all wheels on the train.
- IoT will know the wheel diameter, and therefore its circumference.
- When the difference between RPM and GPS based speeds is greater than 5 mph, send an alert signal and suggest slowing down.
- When the difference between RPM and GPS based speeds is greater than 2 mph, send an inform signal and suggest slowing down.

3.2.5. Detect if ice can be formed on rails and inform the operator

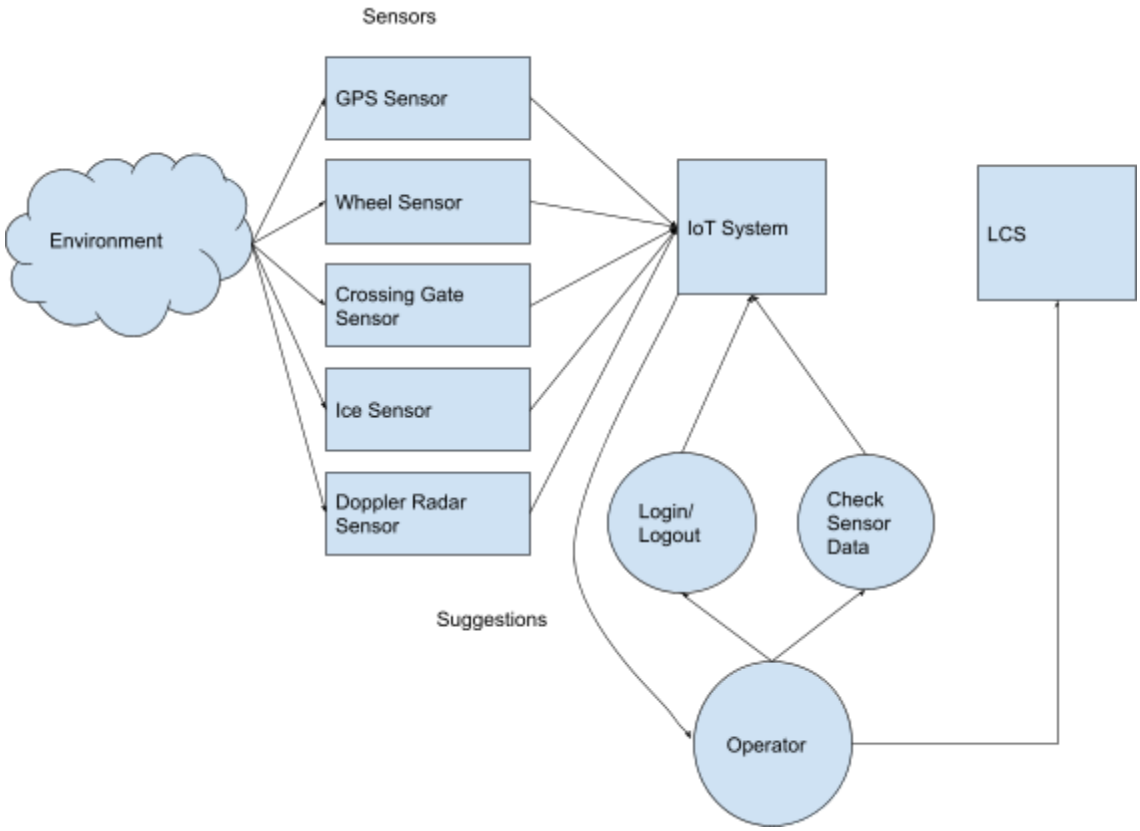
- Will require a temperature and humidity combination sensor.
- Will poll the environment at a location close to the level of the railroad
- Will be able to measure temperature between -30C and +30C
- Will be able to measure humidity between 0% to 100%
- When humidity is detected between 72% and 78% and temperature is detected to be below 0C, send an inform signal.

3.2.6. Be able to handle Inform Signals

- All inform signals will be displayed in yellow text, will include the main details about the message, and a suggestion about further actions
- Inform signals will include: wheel slippage of more than 2 mph but no more than 5 mph, objects behind or in front being within 10 miles of the train, the possibility of ice formation.

3.2.7. Be able to handle Alert Signals

- All alert signals will be displayed in red text, will include the main details about the message, and a suggestion about further actions
- Alert signals will include: wheel slippage of more than 5 mph, objects behind or in front being within 5 miles of the train



4. Analysis Model

4.1. Use Cases

4.1.1. Use Case: User logs into the system

Use Case No.: 4.1.1
Primary Actor: User
Secondary Actor: Control Center
Goal: For the conductor to authenticate themselves and turn on the system
Precondition: The system is waiting for a user
Trigger: Conductor initiates with the system
Scenario:

1. The Control Center asks for credentials
2. The user enters their credentials

3. Control Center checks credentials

4. Once authenticated, the system begins detecting all connected sensors and displays a message of how many and what kinds of sensors have been detected.

5. Control Center will begin listening to the sensors

Exceptions:

The credentials are not valid and will ask for them again.

4.1.2. **Use Case:** User logs out of the system
Use Case No.: 4.1.2
Primary Actor: User
Secondary Actor: Control Center
Goal: For the conductor to log out of the system
Precondition: Control Center is running, and the train is at full stop
Trigger: Conductor presses log out button
Scenario:

1. User hits the logout button

2. Control Center logs the user out and brings up the login screen again

3. Control Center begins to disengage all connected sensors

Exceptions:

1. If the train is not at full stop, the logout button will not be shown as an option

4.1.3. **Use Case:** Technician interacts with logs
Use Case No.: 4.1.3
Primary Actor: Technician
Secondary Actor: Logging System
Goal: Extract, view, or wipe logs recorded on the device
Precondition: The train is stationary, no one is logged into the system
Trigger: Technician logs into the system with technician specific credentials
Scenario:

1. A technician is sent out to extract/view/wipe log files

2. Technician logs into the system and has options to interact with the log files

3. The technician may chose to view, extract, or delete their contents

4.1.4. **Use Case:** GPS Sensor sends speed update
Use Case No.: 4.1.4

Primary Actor: Sensor Array
Secondary Actor: Control Center
Goal: Send data about speed of train based on GPS coordinates
Precondition: The train is moving, GPS sensor is installed, conductor is logged in
Trigger: GPS refreshes its coordinates
Scenario:

1. Sensors periodically updates its coordinates at a set refresh rate.
2. Sensor array calculates speed based on previous cached coordinates and refresh rate.
3. Sensor array sends speed data to Control Center

4.1.5. **Use Case:** Object detected on rails
Use Case No.: 4.1.5
Primary Actor: Sensor Array
Secondary Actor: Control Center
Goal: Send signal about object proximity
Precondition: Train is traveling faster than 10 miles per hour, doppler radar is installed, conductor is logged in
Trigger: Doppler radar sensor detects object in front of train
Scenario:

1. Doppler sensor periodically sends waveforms in front of the train, measuring their wavelength on return
2. Sensor detects returning waveform
3. If returning waveform traveled less than 5 miles send alert signal, otherwise send inform signal to the Control Center

4.1.6. **Use Case:** Control Center detects slippage
Use Case No.: 4.1.6
Primary Actor: Control Center
Secondary Actor: Sensor Array
Goal: Send signal about wheel slippage
Precondition: Train is moving, wheel and GPS sensor are installed, conductor is logged in
Trigger: Difference between wheel distance traveled and GPS distance traveled is between 100 and 600 yards
Scenario:

1. Control Center receives location data from GPS, and rotation data from wheel sensors
2. Control Center calculates current speed according to Wheel and GPS sensors
3. Control Center sends inform signal, if the difference between

them is between 2 to 5 mph. If it is greater than 5 mph, sends alert signal.

4.1.7. **Use Case:** Gate sensor detects gate
Use Case No.: 4.1.7
Primary Actor: Sensor Array
Secondary Actor: Control Center
Goal: Send inform signal that gate is detected
Precondition: Train is moving, gate sensor is installed, conductor is logged in
Trigger: Sensor detects gate
Scenario:

1. Sensor detects gate
2. Sensor sends inform signal along with distance to gate
3. Control Center display message along with additional instructions on blowing the train horn

4.1.8. **Use Case:** Ice sensor detects optimal conditions
Use Case No.: 4.1.8
Primary Actor: Sensor Array
Secondary Actor: Control Center
Goal: Send inform signal that ice formation conditions are optimal
Precondition: Train is moving, ice sensor is installed, conductor is logged in
Trigger: Sensor detects optimal ice formation conditions
Scenario:

1. Ice Sensor detects optimal ice formation conditions
2. Sensor sends inform signal to Control Center

4.1.9. **Use Case:** Control Center logs event
Use Case No.: 4.1.9
Primary Actor: Control Center
Secondary Actor: Logging System
Goal: Log data that is coming in from sensors, conductor is logged in
Precondition: Conductor is logged into the system
Trigger: Data is processed by the Control Center
Scenario:

1. Control Center processes data to form a conclusion
2. Control Center generates event based on conclusion
3. Control Center passes event to Logging System

4.1.10. **Use Case:** Control Center must inform the operator
Use Case No.: 4.1.10

Primary Actor: Control Center
Secondary Actor: Conductor
Goal: Display inform message
Precondition: Sensor array is sending data, conductor is logged in
Trigger: Control Center receives inform signal from Sensor Array
Scenario:

1. Control Center receives inform signal from sensors
2. Control Center processes signal into displayable information
3. Control Center displays the message and recommended actions, if any.

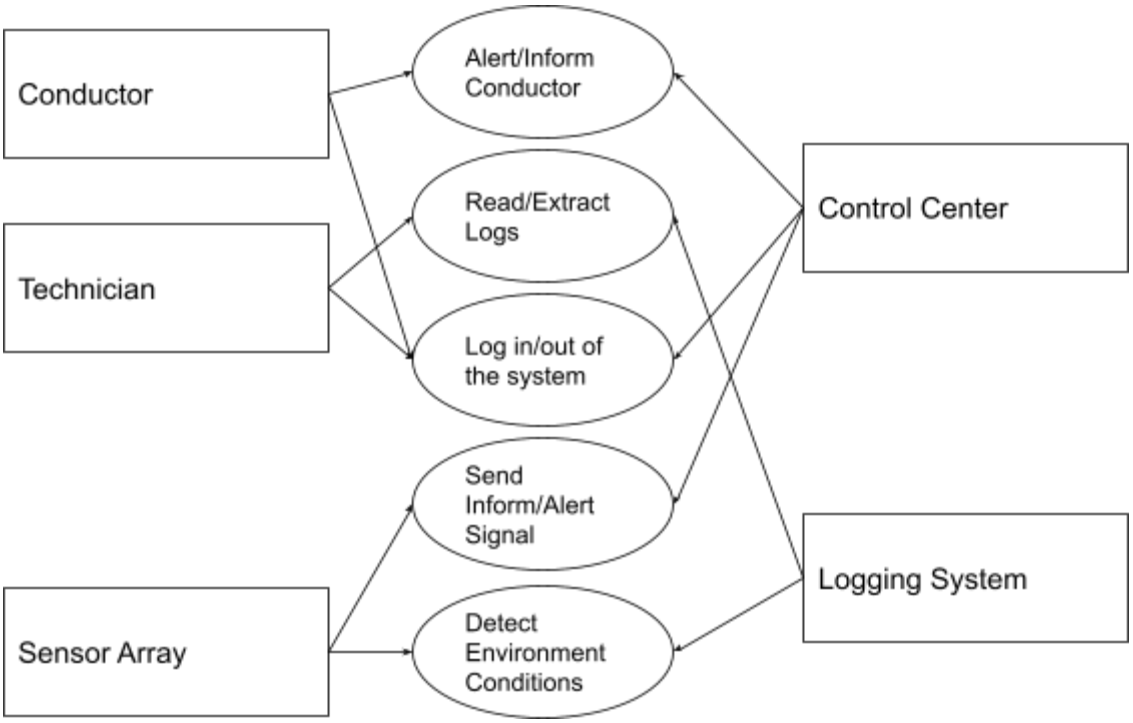
4.1.11. **Use Case:** Control Center must alert the operator
Use Case No.: 4.1.11
Primary Actor: Control Center
Secondary Actor: Conductor
Goal: Display inform message
Precondition: Sensor array is sending data, conductor is logged in
Trigger: Control Center receives alert signal from Sensor Array
Scenario:

1. The Control Center receives alert signal from sensors or processes combined data into an alertable situation
2. The Control Center processes signal into displayable information
3. The Control Center makes a notification sound and displays the message and recommended actions.

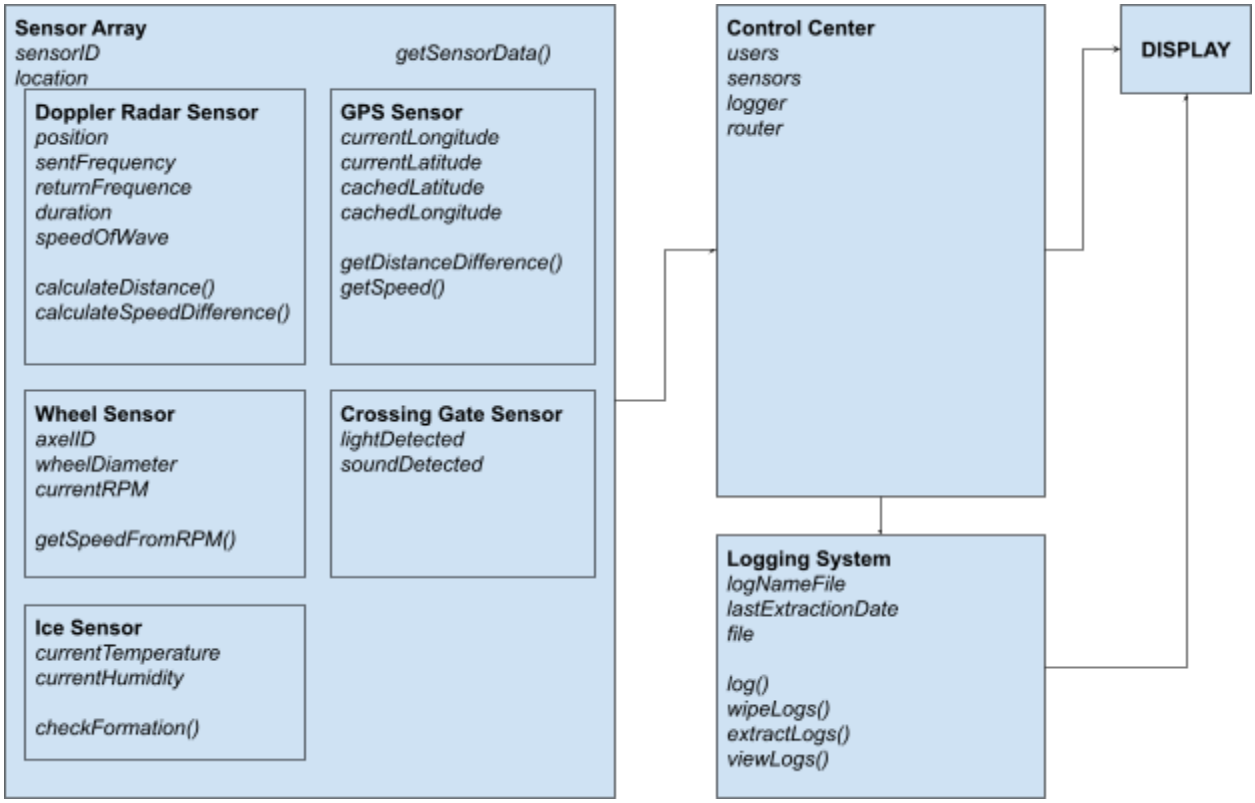
Exceptions:

1. Alert signal is suppressed by override data

4.2. UML Use Case Diagram



UML Class Diagram



Relationships:

-Sensor Array to Control Center: sends data to Control Center for processing after it is collected by the sensor modules

-Control Center to Logging System: sends all processing events that occur within the Control Center to be recorded

-Logging System to Display: displays log files and log file options

-Control Center to Display: displays menu options, generic, inform, and alert signals.

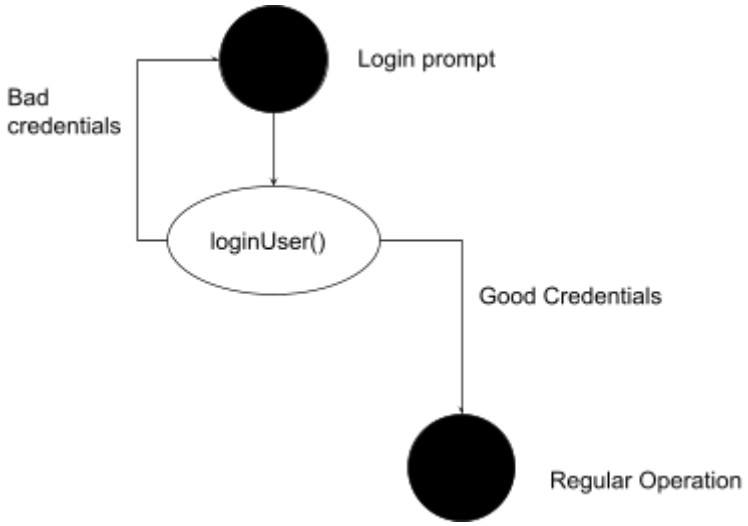
4.3. UML CRC Index Cards

Control Center	**Sensor Array**	**Logging System**
Description: Processes data for the IoT coming in from the sensor array.	**Description**: Ingests inputs from physical sensors and passes them to the CC	**Description**: Records all events happening in the CC, which are passed from the CC
Responsibilities: ● Take data from sensor array ● Process data from sensor array ● Make decisions based on the processed information and display	**Responsibilities**: ● Connect to sensors via sensor router interface ● Parse raw data coming in from sensors via router ● Pass parsed data to Control Center	**Responsibilities**: ● Records events that occur in the Control Center, like informs/alerts, data calculations, data cross checking, and user logins/logouts

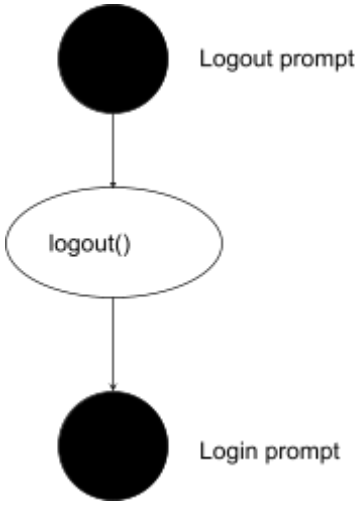
them to the user		• Give access to the saved logs to the technicians upon request
Collaborators: • Conductor/Technician • Sensor Array • Logging System • Display	**Collaborators:** • Sensor Router • Control Center	**Collaborators:** • Technician • Control Center • Display

4.4. UML Activity Diagrams

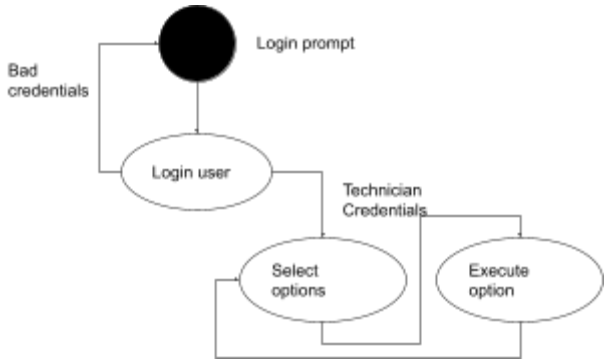
4.4.1. Conductor logs into the system



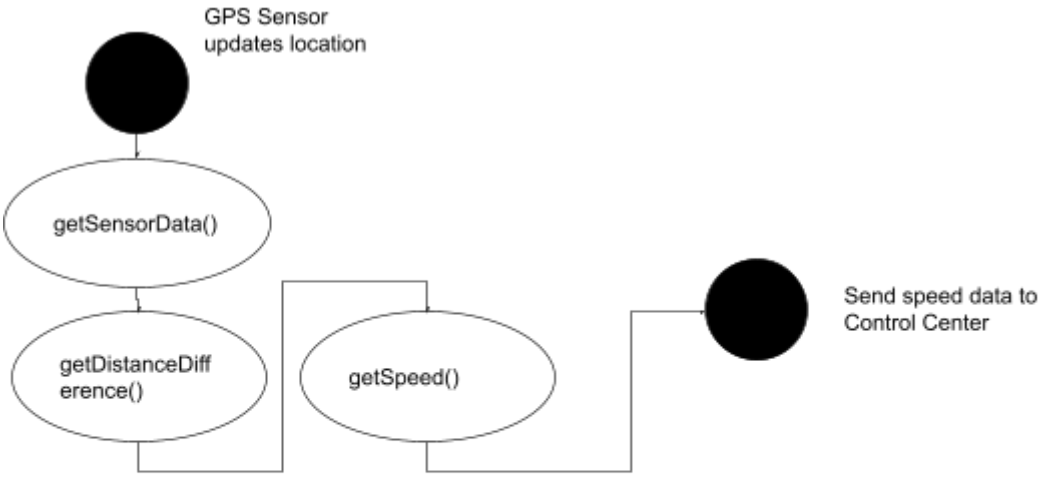
4.4.2. Conductor logs out of the system



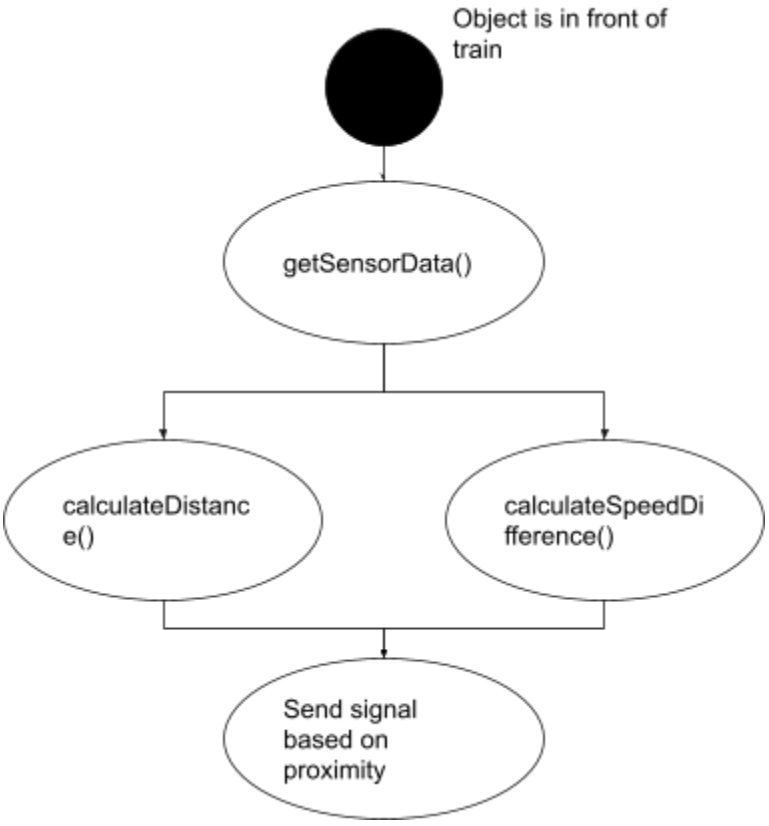
4.4.3. Technician interacts with logs



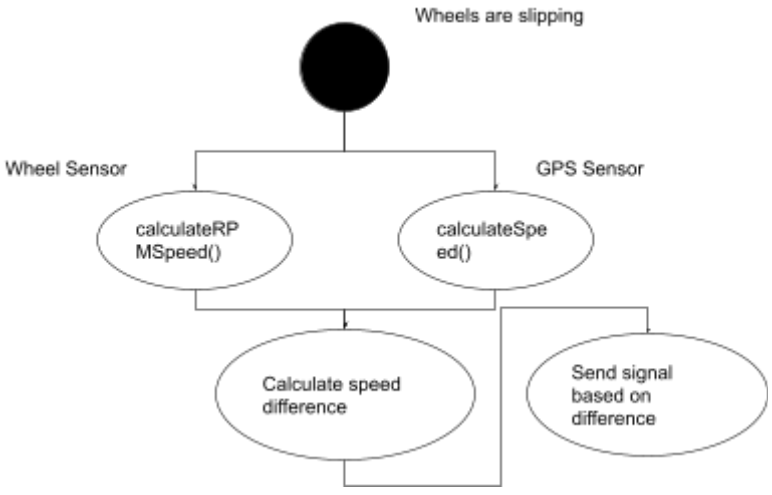
4.4.4. GPS Sensor sends speed update



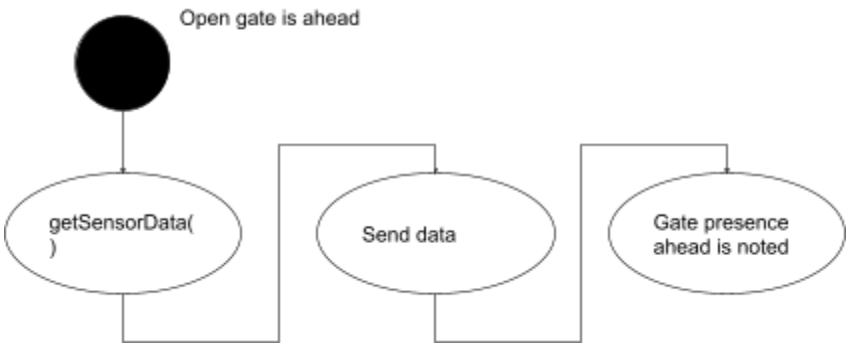
4.4.5. Object is detected on rails



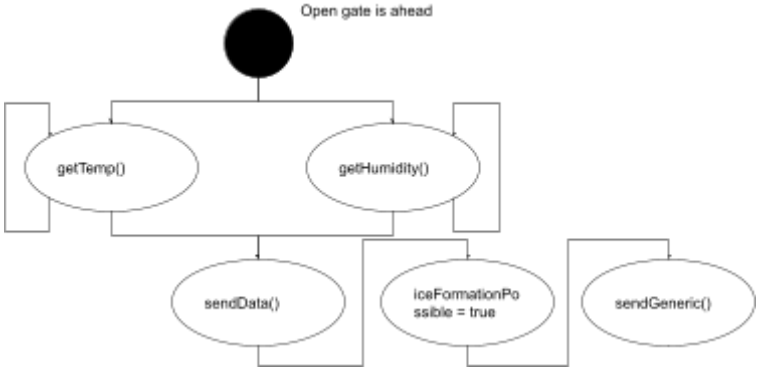
4.4.6. Control Center detects slippage



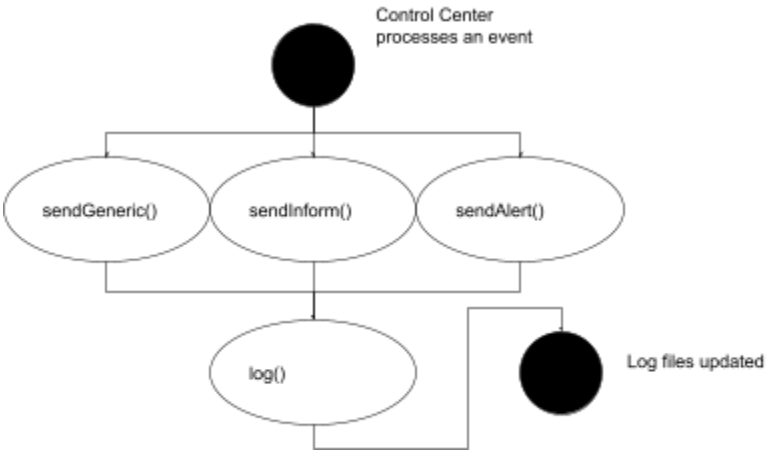
4.4.7. Gate sensor detects gate lights



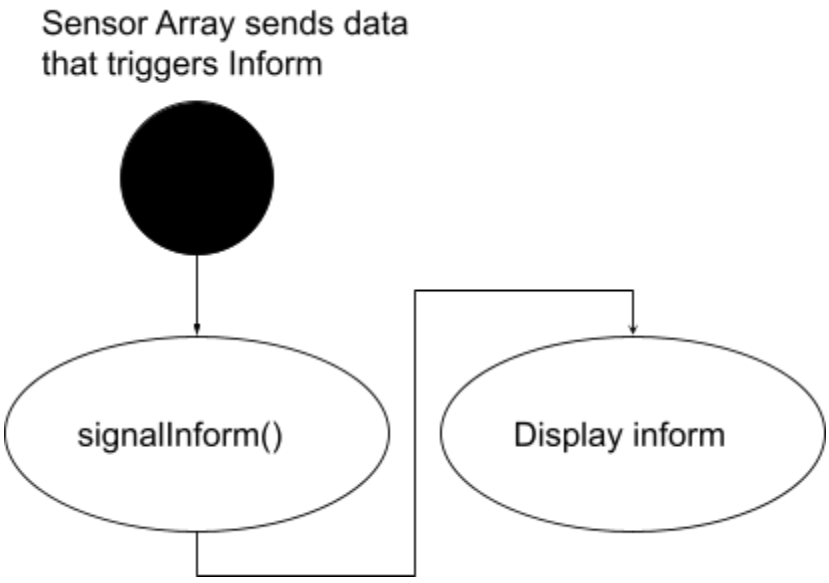
4.4.8. Ice sensor detects optimal conditions



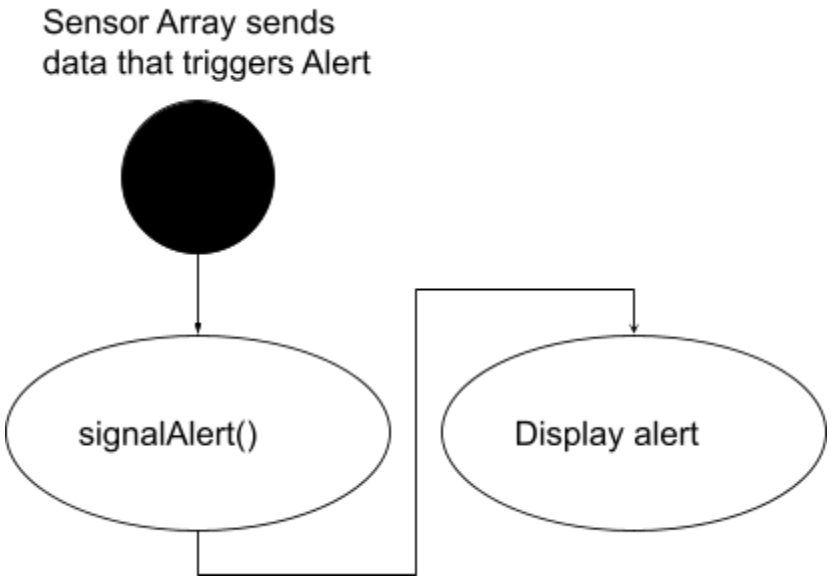
4.4.9. Control Center logs data



4.4.10. Control Center must inform the conductor

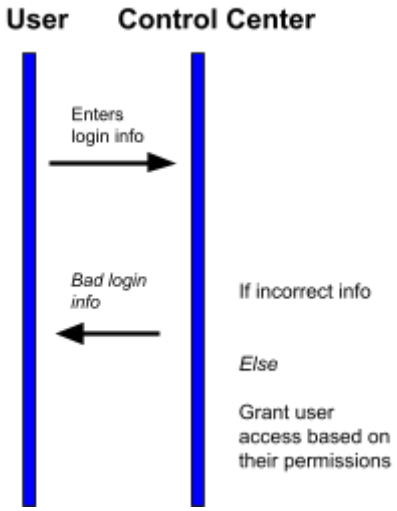


4.4.11. Control Center must alert the operator

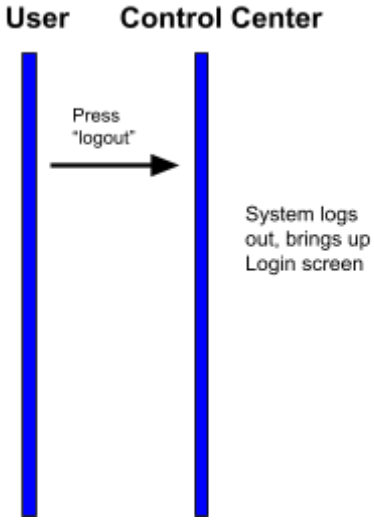


4.5. UML Sequence Diagrams

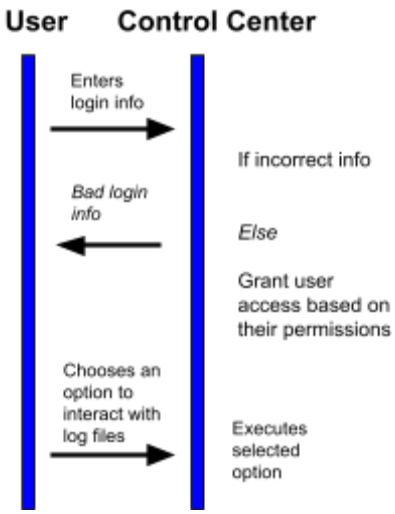
4.5.1. Conductor logs into the system



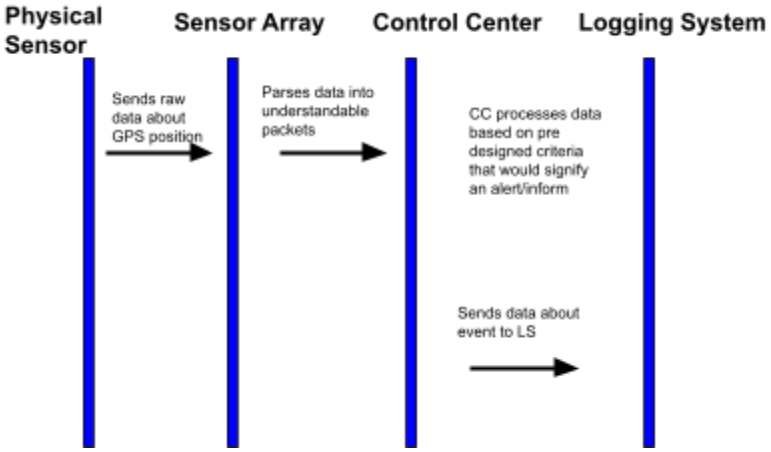
4.5.2. Conductor logs out of the system



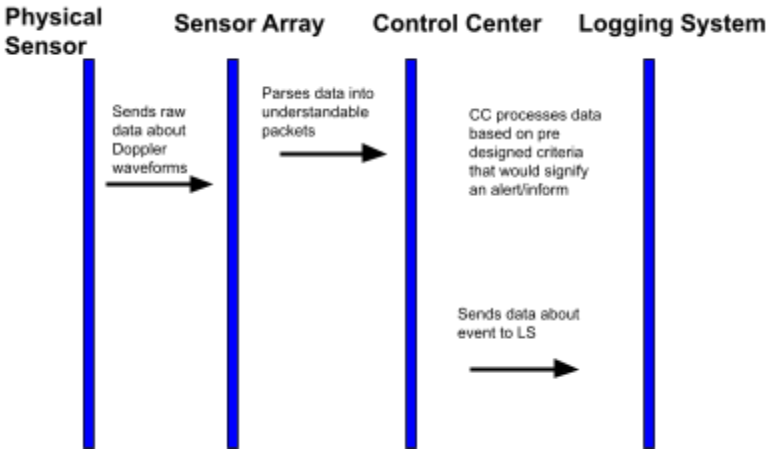
4.5.3. Technician interacts with logs



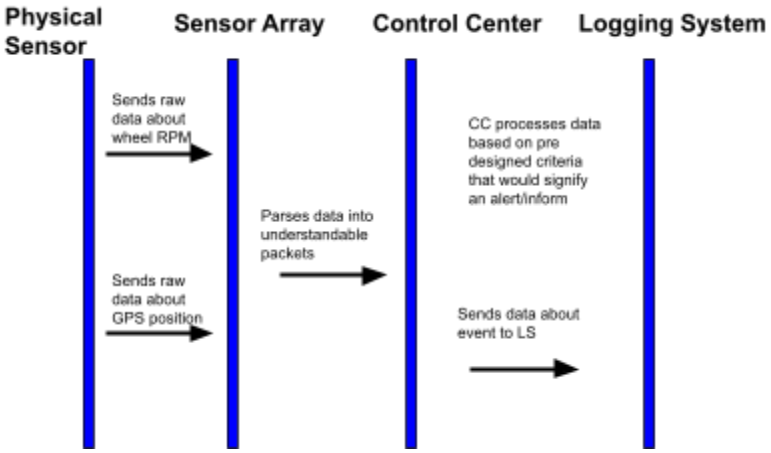
4.5.4. GPS Sensor detects crossing gate



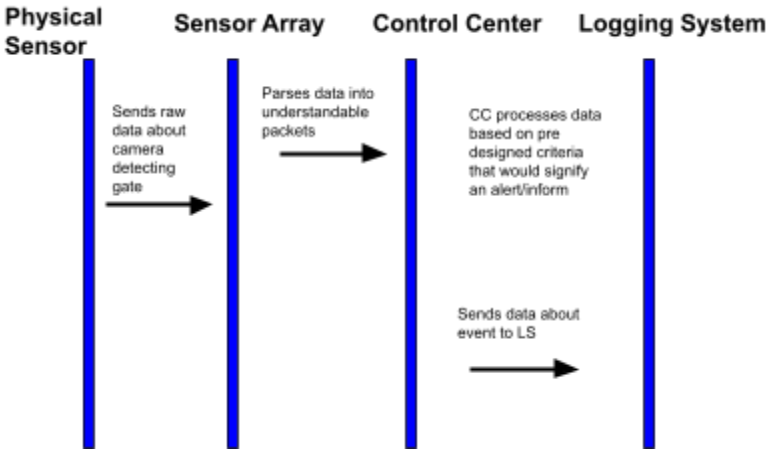
4.5.5. Object is detected on rails



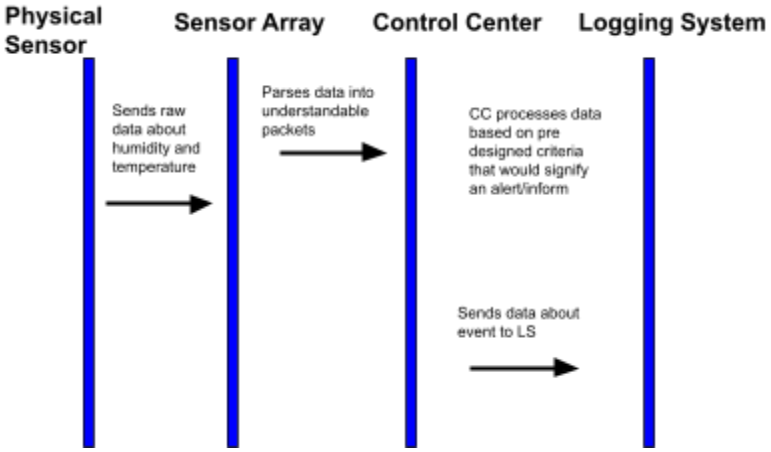
4.5.6. Control Center detects slippage



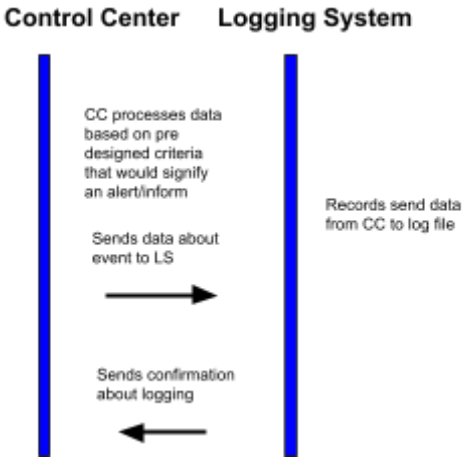
4.5.7. Gate sensor detects gate lights



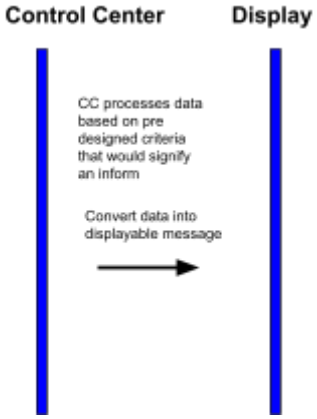
4.5.8. Ice sensor detects optimal conditions



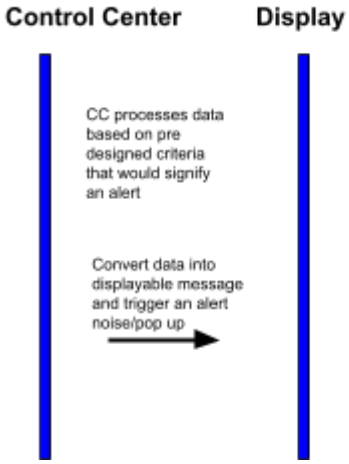
4.5.9. Control Center logs data



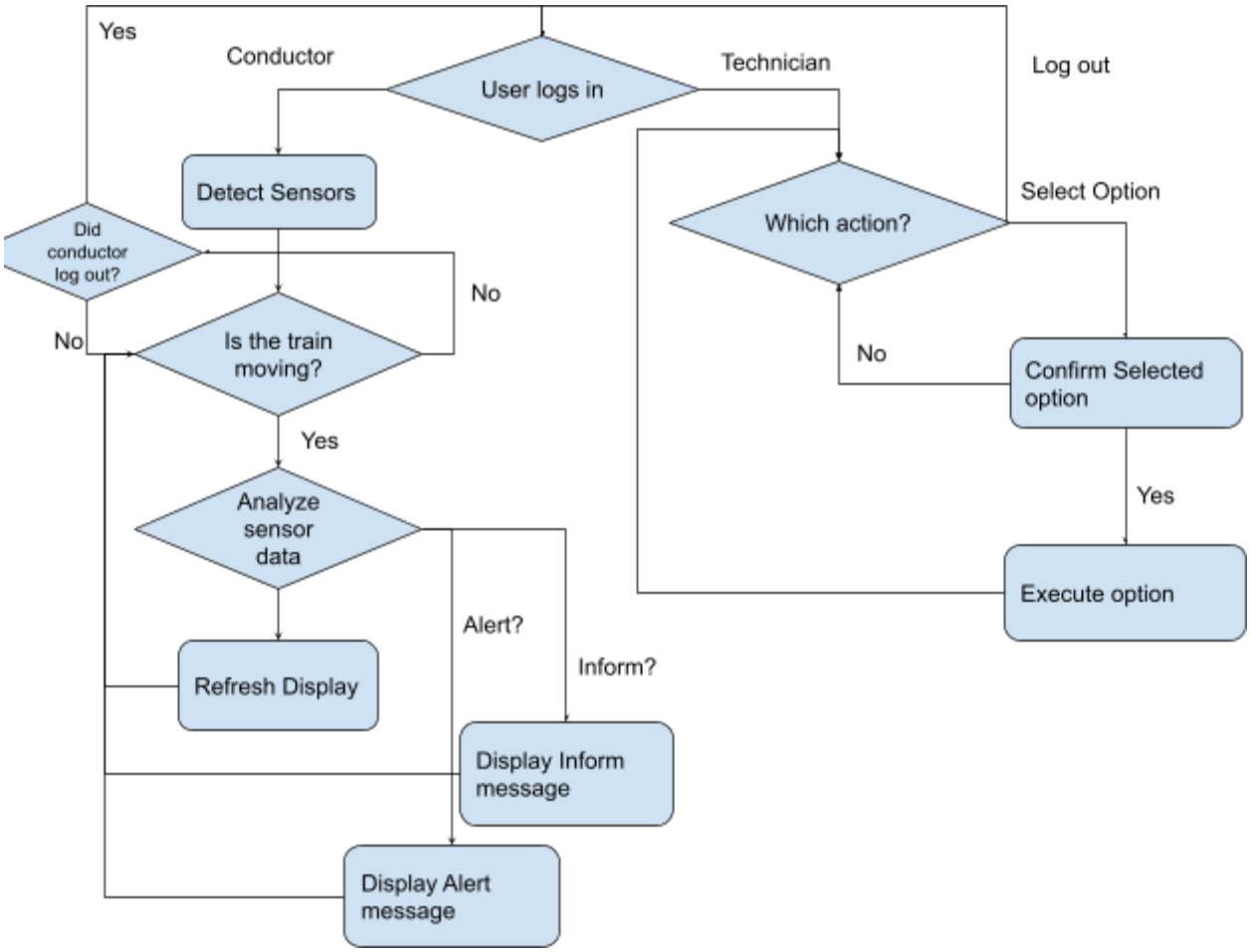
4.5.10. Control Center must inform the conductor



4.5.11. Control Center must alert the operator



4.6. UML State Diagram



5. System Architecture

5.1. Architecture Model Choice

5.1.1. View Controller Architecture

- **Application:** The currently logged in user would view the data presented to it by a component that is responsible for generating viewable data. That data that it will be showing will depend less on the user's input but more on the data supplied by sensors. That data will feed into modules that will process it and generate viewable information.
- **Pros:**
    - More reliance on sensor data than viewer input.
    - More focused on processing data and delivering information to the viewer.
- **Cons:**
    - Less interaction from the viewer.

    - Backend process is unknown to the user.
    - No explicit distribution of data, may cause bottlenecks.

5.1.2. Data Centered Architecture

- **Application:** The data is not the most important aspect of the IoT, it is the way it is processed and the actions that the IoT takes based on it. The logging system has the highest resemblance to a data structure that the system may use, with the user being its only accessor, as the control center never needs to reassess the information logged.
- **Pros:**
    - Works well with data log collection on computer to track each sensors data and trains activity
    - Reflects the use of many sensors to make decisions at central hubs
    - Puts data access at the forefront of the model. Accessing and modifying data stored in a data center is fast and reliable
- **Cons:**
    - Data is not at the core of what IoT needs to be.
    - The technician is not accessing the data in a time sensitive environment and the conductor doesn't need to access it at all.
    - Once the data is logged it is not modifiable as the assumption is that it was parsed correctly to begin with.
    - Isn't as intuitive in terms of user interaction and system responsiveness to data input. IoT is as much about responses to data as it is data collection, and this is not necessarily represented by a data-centered model.

5.1.3. Pipe and Filter Architecture

- **Application:** Piping data from sensors into filters for it to be processed is close to what the IoT should be. The physical sensors supply the data to the sensor array, which sends it to the control center, which parses it and sends it to the screen and logging system.
- **Pros:**
    - One way approach, data can only be input from one direction and cannot be supplied in the opposite way.
    - Makes for a clear understanding of which path what kind of data should take.
    - Shows the way data is adapted as it moves through the system to be readable and usable
    - Reflects the ways in which important or urgent information is filtered and responded to accordingly
- **Cons:**

    - If there aren't enough pipes/filters or the data on the input is too similar, may cause bottlenecks.
    - Having some pipes/filters barely being used with others not being able to process the data in time.
    - Not necessarily clear in regards to the user's interaction with the system.

5.1.4. Call Return Architecture

- **Application:** Data flows into one module which is then sent off to a specific, more specialized module(s) that processes the data and sends a reply back to the parent module.
- **Pros:**
    - Hierarchical system with the top most modules having to be more robust, generic, and capable of sending and receiving data.
    - Lower, more specialized modules, can be made to process less data as they are going to be called less often.
    - Reflects the way in which programs return to the central hub machine, affecting decisions and alerts made, and what is presented to the user
    - Shows the web of information presented by the sensors
- **Cons:**
    - May cause bottlenecks with the amount of data that the sensors may supply if everything has to go through a shared module.
    - Puts a lot of structural pressure on the central machine and early branches from that tree: if one fails, many may fail
    - Data needs to travel double distance and may slow down the responsiveness.

5.1.5. Object Oriented Architecture

- **Application:** Each type of sensor may want to communicate with its own specialized object that will know how to process its type of data. These objects would all communicate with the controller objects that will have to decide on what type of data to display and log.
- **Pros:**
    - Simplified flow of data.
    - Cohesive way for modules to come together to accomplish a task.
    - Intuitive architecture with regards to the fact that there are independent sensors, which can fall within different classes.
- **Cons:**
    - Doesn't clearly reflect the centrality of the console as compared to some other structures

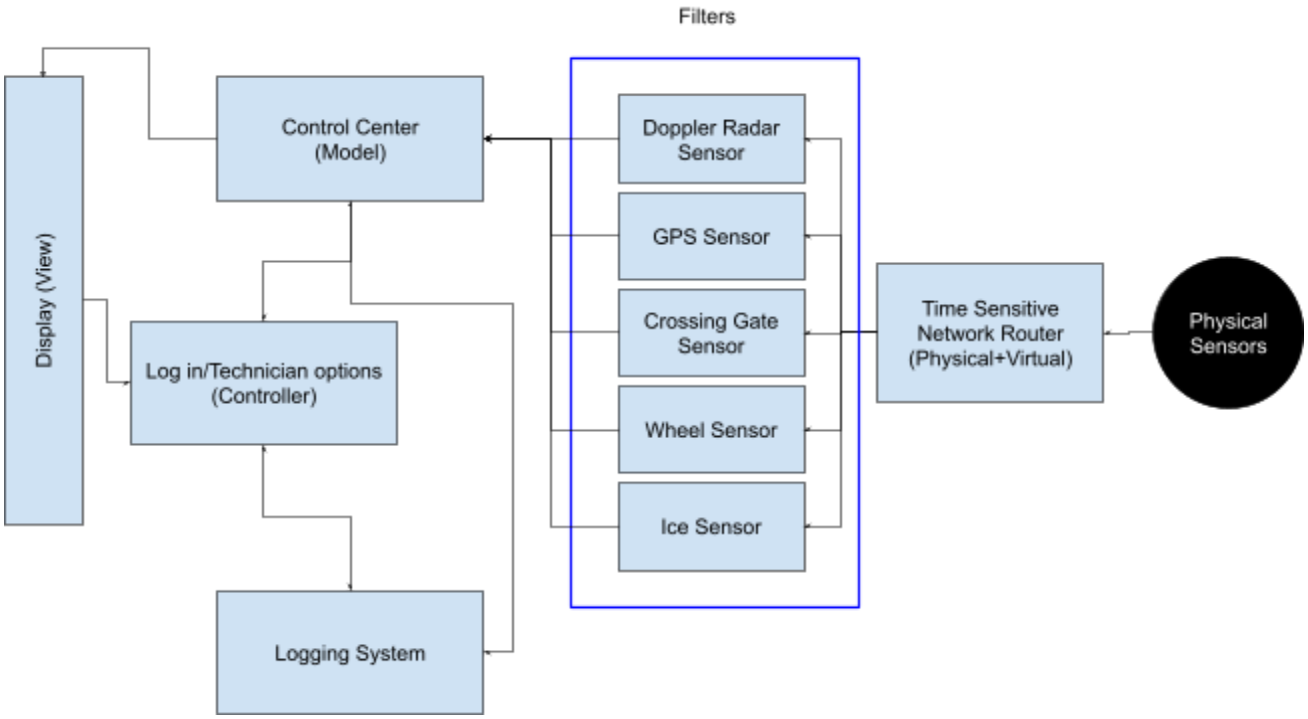
    - Doesn't clearly show the movement and processing and filtering of data
    - Either the architecture requires less objects with each one being more general and capable of handling more tasks
    - Or more objects at the price of more specialization but less capabilities.

5.1.6. Layered Architecture

- **Application:** The IoT would have to inject sensor data into the center most layer and have it travel up through the more and more complex layers until it becomes viewable information.
- **Pros:**
    - Reflects the operator's different levels of visibility on the central hub
    - Each layer is mostly isolated and works more closely with components that are found on its own layer (like sensor data converters or display information generators).
- **Cons:**
    - Data will not travel lower into the layers, only upwards.
    - The communication between the layers will be minimal.
    - The amount of layers may not be sufficient enough to justify applying the layered model.
    - The traversal of layers may slow down the system

5.1.7. Conclusion: View Controller Architecture

- The chosen architecture for the IoT system will be the Object Oriented Structure with elements borrowed from Viewer-Controller and Pipe-Filter models as well. The most important aspect of the system is to deliver accurate information to the user on time. The way that data will be processed is hidden from the user and does not need any user input to function. When a user does need to interact with the system (like in the case of a technician), the system won't be processing any data from the sensors. The amount of data to be processed from the sensor may be overwhelming but a simplified pipe-filter model can be employed for data intake and processing.



Physical sensors will all be connected to the Signal Router which will be connected to the IoT. Using the virtual router interface, the IoT will be able to pipe the signal through sensor filters that will pipe the information to the Control Center which will process the data and make decisions based on it and log events to the Logging System. User interaction is minimal so the controller will consist of user login and the options that a technician-type user has (to interact with the Logging System). Processed data from the Control Center will be piped to the display. The display will have either a login screen, technician options, or sensor data updates. Data updates will consist of general information like speed, time running, coordinates according to the GPS, and outside temperature/humidity. It will also include inform and alert signals that will be raised by the Control Center.

6. Code

IoTSimulator.java

```
import javax.swing.JOptionPane;
import java.util.*;
import java.text.DecimalFormat;
import java.io.FileNotFoundException;
import java.io.File;

public class IoTSimulator {
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_RED = "\u001B[31m";
    public static final String ANSI_GREEN = "\u001B[32m";
```

```
    public static final String ANSI_YELLOW = "\u001B[33m";
    public static final String ANSI_WHITE = "\u001B[37m";
    public static final String ANSI_BLUE = "\u001B[34m";

    private static DecimalFormat df = new DecimalFormat("0.0");

    private static ArrayList<String[]> users = new ArrayList<String[]>();

    public static void infoBox(String infoMessage, String titleBar)
    {
        JOptionPane.showMessageDialog(null, infoMessage, "InfoBox: " + titleBar,
JOptionPane.INFORMATION_MESSAGE);
    }

    static Dictionary<String, Sensor> sensors = new Hashtable<String, Sensor>();

    static boolean isCond = true;
    static boolean loggedIn = false;

    private static LoggingSystem logger = new LoggingSystem("log.hrt");
    private static RouterSim router;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try { // open list of users file
            File myObj = new File("users.hrt");
            Scanner myReader = new Scanner(myObj);
            while (myReader.hasNextLine()) {
                String data = myReader.nextLine();
                String[] userlog = data.split(" ", 4);
                users.add(userlog);
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("Could not detect local user list");
            e.printStackTrace();
        }
        System.out.print(ANSI_WHITE);
        while (true){ // main login while-true loop
            System.out.println(ANSI_GREEN + "-------Hugs the Rail Onboard Sensor
Console--------");
            System.out.print(ANSI_WHITE);
            System.out.print("Username: "); // ask for username
            System.out.print(ANSI_GREEN);
```

```
            String user = sc.next();
            if (user.equals("quit")){ loggedIn=false; return; } // if username == quit, quit program
            System.out.print(ANSI_WHITE);
            System.out.print("Password: "); // ask for password
            System.out.print(ANSI_GREEN);
            String pswd = sc.next();
            for (int x = 0; x < users.size(); x++){ // search user file for valid credentials
                if (users.get(x)[0].equals(user) && users.get(x)[1].equals(pswd)){
                    if (users.get(x)[2].equals("cond")){
                        isCond = true;
                        System.out.println("---Welcome Conductor " + users.get(x)[3] + "---"); // greet with
preferred name
                        logger.log(user, "Conductor logged in");
                        loggedIn = true;
                    } else if (users.get(x)[2].equals("tech")){
                        isCond = false;
                        System.out.println("---Welcome Technician " + users.get(x)[3] + "---"); // greet
with oreferred name
                        logger.log(user, "Technician logged in");
                        loggedIn = true;
                    }
                }
            }

            if (!loggedIn){ // no valid user found, return to login loop
                System.out.println(ANSI_YELLOW + "Username/password combination not found in
local database.\nContact a technician to rectify this problem." + ANSI_WHITE);
                logger.log("IOTSYS", "Incorrect login attempt");
            } else {
                if (isCond) { // conductor branch
                    System.out.print(ANSI_WHITE);
                    System.out.println("Reading Global Sensor List...");
                    logger.log("IOTSYS", "Sensor list initialized");
                    int numGPS = 0, numDRS = 0, numIFS = 0, numCGS = 0, numWS = 0;
                    try {
                        Thread.sleep(1400);
                    } catch (InterruptedException e) {}
                    System.out.print(ANSI_YELLOW);
                    try { // lookup sensor list
                        File myObj = new File("sensors.hrt");
                        Scanner myReader = new Scanner(myObj);
                        while (myReader.hasNextLine()) {
                            String data = myReader.nextLine();
                            System.out.println(data);
```

```
                    String[] sensorLine = data.split(" ");
                    String ID = sensorLine[0];
                    if (ID.equals("WS")){ // ID match what sensors are found
                        sensors.put(sensorLine[1], new WheelSensor(sensorLine[1], sensorLine[2],
sensorLine[3], Float.parseFloat(sensorLine[4])));
                        logger.log("IOTSYS", ID + " " + sensorLine[1] + " Wheel Sensor detected");
                        numWS++;
                    } else if (ID.equals("GPSS")){
                        sensors.put(sensorLine[1], new GPSSensor(sensorLine[1], sensorLine[2]));
                        logger.log("IOTSYS", ID + " " + sensorLine[1] + " GPS Sensor detected");
                        numGPS++;
                    } else if (ID.equals("CGS")){
                        sensors.put(sensorLine[1], new CrossingGateSensor(sensorLine[1],
sensorLine[2]));
                        logger.log("IOTSYS", ID + " " + sensorLine[1] + " Crossing Gate Sensor
detected");
                        numCGS++;
                    } else if (ID.equals("IFS")){
                        sensors.put(sensorLine[1], new IceFormationSensor(sensorLine[1],
sensorLine[2]));
                        logger.log("IOTSYS", ID + " " + sensorLine[1] + " Ice Formation Sensor
detected");
                        numIFS++;
                    } else if (ID.equals("DRS")){
                        sensors.put(sensorLine[1], new DopplerRadarSensor(sensorLine[1],
sensorLine[2], Integer.parseInt(sensorLine[3]), Double.parseDouble(sensorLine[4])));
                        logger.log("IOTSYS", ID + " " + sensorLine[1] + " Doppler Radar Sensor
detected");
                        numDRS++;
                    }
                }
                myReader.close();
            } catch (FileNotFoundException e) { // sensor list not found, must restart
                System.out.print(ANSI_WHITE);
                System.out.println("Sensor list not found. Provide a sensor list and restart");
                logger.log("IOTSYS", "Sensor list not found");
                return;
            }
            System.out.print(ANSI_GREEN);
            System.out.println("---Sensors Detected Onboard---" + ANSI_WHITE); // list
detected sensors
            System.out.print("GPS Sensors:\t\t");
            if (numGPS<1) {System.out.print(ANSI_YELLOW);}
            System.out.println(numGPS + ANSI_WHITE);
```

```
                System.out.print("Doppler Radar Sensors:\t");
                if (numDRS<2) {System.out.print(ANSI_YELLOW);}
                System.out.println(numDRS + ANSI_WHITE);
                System.out.print("Ice Formation Sensors:\t");
                if (numIFS<1) {System.out.print(ANSI_YELLOW);}
                System.out.println(numIFS + ANSI_WHITE);
                System.out.print("Crossing Gate Sensors:\t");
                if (numCGS<1) {System.out.print(ANSI_YELLOW);}
                System.out.println(numCGS + ANSI_WHITE);
                System.out.print("Wheel Sensors:\t\t");
                if (numWS<4) {System.out.print(ANSI_YELLOW);}
                System.out.println(numWS + ANSI_WHITE);

                if (numGPS>=1 && numDRS>=2 && numIFS>=1 && numCGS>=1 &&
numWS>=4){ // count if enough sensors are onboard
                    System.out.println(ANSI_GREEN + "Minimum amount of sensors met!" +
ANSI_WHITE);
                } else {
                    System.out.println(ANSI_YELLOW + "Warning! " + ANSI_WHITE + "Not enough
sensors detected to meet minimum requirements");
                }
                logger.log("IOTSYS", "Sensors detected");

                System.out.print(ANSI_YELLOW + "Begin simulated ride? [y/n] " + ANSI_GREEN);
// confirmation to begin script
                String response = sc.next();
                if (!(response.equals("y") || response.equals("Y"))){ // confirmation check
                    loggedIn = false;
                    break;
                }

                System.out.println(ANSI_YELLOW + "---Start of testing script---" + ANSI_WHITE);
                router = new RouterSim("script.hrt", 1500,
numGPS+numCGS+numDRS+numIFS+numWS); // create router object

                double secondsOfHorn = 0;
                while (true) { // loop for reading sensor data from router
                    String[] block = router.getDataBlock(); // block of sensor data for each sensor
                    if (block == null){
                        System.out.println(ANSI_YELLOW + "---End of testing script---" +
ANSI_WHITE);
                        loggedIn = false;
                        logger.log(user, "Conductor logs out");
                        break;
```

```
        } else {
            ArrayList<Double> speedFromRPM = new ArrayList<Double>();
            ArrayList<Double> speedFromGPS = new ArrayList<Double>();
            ArrayList<Double> temperature = new ArrayList<Double>();
            ArrayList<Double> humidity = new ArrayList<Double>();

            ArrayList<Double> frontDistance = new ArrayList<Double>();
            ArrayList<Double> backDistance = new ArrayList<Double>();
            ArrayList<Double> frontSpeed = new ArrayList<Double>();
            ArrayList<Double> backSpeed = new ArrayList<Double>();

            ArrayList<Double> distanceTo = new ArrayList<Double>();

            boolean lightDetected = false, soundDetected = false, icePossible = false;
            double wheelSlippageSpeedDiff = 0;

            for (int i = 0; i<block.length; i++){ // keep looping for sensor data blocks
                String[] splitblock = block[i].split(" ");
                Sensor s = sensors.get(splitblock[0]); // do action based on type of sensor
                if (s instanceof WheelSensor){ // WheelSensor
                    WheelSensor ws = (WheelSensor) s;
                    ws.getSensorData(block[i]);
                    speedFromRPM.add(ws.getSpeedFromRPM());
                } else if (s instanceof CrossingGateSensor){ // CrossingGateSensor
                    CrossingGateSensor cgs = (CrossingGateSensor) s;
                    cgs.getSensorData(block[i]);
                    lightDetected = lightDetected || cgs.getLightDetected();
                    soundDetected = soundDetected || cgs.getSoundDetected();
                    distanceTo.add(cgs.getDistanceTo());
                } else if (s instanceof GPSSensor) { // GPSSensor
                    GPSSensor gs = (GPSSensor) s;
                    gs.getSensorData(block[i]);
                    speedFromGPS.add(gs.getSpeed());
                } else if (s instanceof IceFormationSensor){ // IceFormationSensor
                    IceFormationSensor ifs = (IceFormationSensor) s;
                    ifs.getSensorData(block[i]);
                    temperature.add(ifs.getTemperature());
                    humidity.add(ifs.getHumidity());
                    icePossible = icePossible || ifs.checkFormation(); // if one sensor detects
conditions
                } else if (s instanceof DopplerRadarSensor){ // DopplerRadarSensor
                    DopplerRadarSensor drs = (DopplerRadarSensor) s;
                    drs.getSensorData(block[i]);
                    if (drs.getPosition() == 1){ // data pertaining to front DRS
```

```
                        if (drs.getReturnFrequency() > 0) { // only add data from sensors that
detected something
                            frontDistance.add(drs.calculateDistance());
                            frontSpeed.add(drs.calculateSpeedDifference());
                        }
                    } else { // data pertaining to back DRS
                        if (drs.getReturnFrequency() > 0) {
                            backDistance.add(drs.calculateDistance());
                            backSpeed.add(drs.calculateSpeedDifference());
                        }
                    }
                }
                //print generic information update
                System.out.println(ANSI_GREEN +
"_____");
                System.out.println(ANSI_WHITE + "Speed:= " + ANSI_GREEN +
df.format(findMean(speedFromRPM)) + ANSI_WHITE + " mph (RPM) " + ANSI_GREEN +
df.format(findMean(speedFromGPS)) + ANSI_WHITE + " mph (GPS)");
                System.out.println("Temperature:= " + ANSI_GREEN +
df.format(findMean(temperature)) + "\u00B0C" + ANSI_WHITE + " Humidity:= " + ANSI_GREEN
+ df.format(findMean(humidity)) + "%" + ANSI_WHITE);

                logger.log("GNRS", df.format(findMean(speedFromRPM)) + " (RPM); " +
df.format(findMean(speedFromGPS)) + " (GPS); T: " + df.format(findMean(temperature)) + " C;
H: " + df.format(findMean(humidity)) + "%");

                if (findMean(speedFromRPM)>5){ // check if train is moving fast, RPM is more
accurate for slower speeds
                    if (!frontDistance.isEmpty()) { // object detected ahead
                        if (findMean(frontDistance) <= 5){ // less than 5 miles
                            System.out.println(ANSI_RED + "[ALERT] Object ahead! " +
df.format(findMean(frontDistance)) + " mi. going " + df.format(findMean(speedFromGPS) +
findMean(frontSpeed)) + " mph" + ANSI_WHITE);
                            System.out.println(ANSI_RED + ">>> Signal with horn and brake" +
ANSI_WHITE);

                            logger.log("ALERT", "Object ahead; " +
df.format(findMean(frontDistance)) + " mi.; " + df.format(findMean(speedFromGPS) +
findMean(frontSpeed)) + " mph");
                        } else if (findMean(frontDistance) <= 10){ //less than 10 miles
                            System.out.println(ANSI_YELLOW + "[INFO] Object ahead! " +
df.format(findMean(frontDistance)) + " mi. ahead at " + df.format(findMean(speedFromGPS) +
findMean(frontSpeed)) + " mph" + ANSI_WHITE);
```

```
                        System.out.println(ANSI_YELLOW + ">>>Slow down by " +
df.format(Math.abs(findMean(frontSpeed))) + " mph" + ANSI_WHITE);

                        logger.log("INFO", "Object ahead; " +
df.format(findMean(frontDistance)) + " mi.; " + df.format(findMean(speedFromGPS) +
findMean(frontSpeed)) + " mph");
                    }
                }
                if (!backDistance.isEmpty()){ // object detected behind
                    if (findMean(backDistance) <= 5){ // less than 5 miles
                        System.out.println(ANSI_RED + "[ALERT] Object behind! " +
df.format(findMean(backDistance)) + " mi. going " + df.format(findMean(speedFromGPS) +
findMean(backSpeed)) + " mph" + ANSI_WHITE);
                        System.out.println(ANSI_RED + ">>>Signal with horn and speed up" +
ANSI_WHITE);

                        logger.log("ALERT", "Object behind; " +
df.format(findMean(backDistance)) + " mi.; " + df.format(findMean(speedFromGPS) +
findMean(backSpeed)) + " mph");
                    } else if (findMean(backDistance) <= 10){ //less than 10 miles
                        System.out.println(ANSI_YELLOW + "[INFO] Object behind! " +
df.format(findMean(backDistance)) + " mi. going " + df.format(findMean(speedFromGPS) +
findMean(backSpeed)) + " mph" + ANSI_WHITE);
                        System.out.println(ANSI_YELLOW + ">>>Speed up by at least " +
df.format(Math.abs(findMean(backSpeed))) + " mph" + ANSI_WHITE);

                        logger.log("INFO", "Object behind; " +
df.format(findMean(backDistance)) + " mi.; " + df.format(findMean(speedFromGPS) +
findMean(backSpeed)) + " mph");
                    }
                }

                if (soundDetected && lightDetected){ // crossing gate detected
                    if (findMean(distanceTo) <= 3.0 && findMean(distanceTo) > 1.5){ // be
aware of gate 3 miles ahead
                        System.out.println(ANSI_YELLOW + "[INFO] Crossing gate ahead (" +
df.format(findMean(distanceTo)) + " mi.)" + ANSI_WHITE);
                        System.out.println(ANSI_YELLOW + ">>>Visually confirm gate" +
ANSI_WHITE);
                    } else if (findMean(distanceTo) <= 1.0 && findMean(distanceTo) >= 0.1){ //
blow horn for 15 sec
                        System.out.println(ANSI_YELLOW + "[INFO] Crossing gate close (" +
df.format(findMean(distanceTo)) + " mi.)" + ANSI_WHITE);
```

```
                        if (secondsOfHorn <= 0 && findMean(distanceTo) >= .9){ // do not
overrite time left if there is time left1
                            secondsOfHorn = 15.0;
                        }
                    } else { // blow horn for 5 sec
                        System.out.println(ANSI_YELLOW + "[INFO] Passing crossing gate ("
+ df.format(findMean(distanceTo)) + " mi.)"  + ANSI_WHITE);
                        if (secondsOfHorn <= 0){ // do not overrite time left if there is time left1
                            secondsOfHorn = 5.0;
                        }
                    }
                    logger.log("INFO", "Crossing gate detected " +
df.format(findMean(distanceTo)));
                }

                if (secondsOfHorn > 0){ // check how much longer to blow horn
                    System.out.println(ANSI_YELLOW + ">>>Blow Horn (" + secondsOfHorn
+ " sec)" + ANSI_WHITE);
                    secondsOfHorn -= router.getRefreshRate()/1000.0;
                }

                if (icePossible){ // ice can form on tracks
                    System.out.println(ANSI_YELLOW + "[INFO] Ice formation possible" +
ANSI_WHITE);
                    System.out.println(ANSI_YELLOW + ">>>Slow down and increase
traction" + ANSI_WHITE);
                    logger.log("INFO", "Ice formation possible");
                }

                if (Math.abs(findMean(speedFromGPS) - findMean(speedFromRPM)) > 1) {
// wheel slippage
                    if (Math.abs(findMean(speedFromGPS) - findMean(speedFromRPM)) >
5){ // major wheel slippage
                        System.out.println(ANSI_RED + "[ALERT] Major wheel slippage");
                        System.out.println(">>>Slow down" + ANSI_WHITE);
                        logger.log("ALERT", "Major wheel slippage: " +
df.format(Math.abs(findMean(speedFromGPS) - findMean(speedFromRPM))));
                    } else if (Math.abs(findMean(speedFromGPS) -
findMean(speedFromRPM)) > 2){ // minor wheel slippage
                        System.out.println(ANSI_YELLOW + "[INFO] Minor wheel slippage");
                        System.out.println(">>>Slow down" + ANSI_WHITE);
                        logger.log("INFO", "Minor wheel slippage: " +
df.format(Math.abs(findMean(speedFromGPS) - findMean(speedFromRPM))));
                    }
```

```
                    }
                }
            }
            try {
                Thread.sleep(router.getRefreshRate());
            } catch (InterruptedException E) {
                System.out.println("Couldn't sleep");
                return;
            }
        }
        loggedIn = false;
    } else { // technician branch
        while (true){ // technician menu loop
            System.out.print(ANSI_WHITE); // list menu options
            System.out.println("Technician menu:\n[$view]\t\t- View Log File\n[$copy]\t\t-
Copy Log File to Directory\n[$wipe]\t\t- Remove Log Data\n[$logout]\t- Log Out");
            System.out.print(ANSI_GREEN);
            System.out.print("$");
            String input = sc.next();
            if (input.equals("logout")){ // technician log out
                logger.log(user, "Technician logs out");
                loggedIn = false;
                break;
            } else if (input.equals("view")) { // print all lines in log file
                logger.log(user, "Technician views log file");
                System.out.print(ANSI_YELLOW);
                logger.viewLogs();
                System.out.print(ANSI_WHITE);
            } else if (input.equals("copy")) { // copy log file to specified location
                System.out.print(ANSI_WHITE + "Directory to copy log file to: " +
ANSI_GREEN);
                String newlogdir = sc.next();
                System.out.print(ANSI_RED + "Copy log file? [y/n] " + ANSI_GREEN);
                String response = sc.next();
                if (response.equals("y") || response.equals("Y")){ // confirmation check
                    if (logger.extractLogs(newlogdir)){
                        System.out.println("Successfully copied log file to " + newlogdir);
                        logger.log(user, "Technician copied log file to " + newlogdir);
                    } else {
                        logger.log(user, "Technician attemped to copy log file to " + newlogdir);
                    }
                }
            } else if (input.equals("wipe")) { // remove data from current log file
                System.out.print(ANSI_RED + "Wipe log file? [y/n] " + ANSI_GREEN);
```

```
                    String response = sc.next();
                    if (response.equals("y") || response.equals("Y")){ // confirmation check
                        logger.log(user , "Technician wiped log file");
                        logger.wipeLogs();
                        logger.log(user , "Technician wiped log file");
                    }
                } else { // unknown command
                    System.out.println(ANSI_GREEN + "$" + input + ANSI_WHITE + " - Command
not recognized");
                }
            }
        }
    }
}

private static double findMode(ArrayList<Double> dob){
    double maxValue = 0;
    int maxCount = 0, i, j;

    for (i = 0; i < dob.size(); ++i) {
        int count = 0;
        for (j = 0; j < dob.size(); ++j) {
            if (dob.get(j) == dob.get(i))
            ++count;
        }

        if (count > maxCount) {
            maxCount = count;
            maxValue = dob.get(i);
        }
    }
    return maxValue;
}

private static double findMean(ArrayList<Double> dob){
    double out = 0;
    for (int i = 0; i<dob.size(); i++){
        out += dob.get(i);
    }
    return out / dob.size();
}

private static double findMedian(List<Double> data) {
```

```
        if (data.size() % 2 == 0)
            return (data.get(data.size() / 2) + data.get(data.size() / 2 - 1)) / 2;
        else
            return data.get(data.size() / 2);
    }
}
```

LoggingSystem.java

```
import java.util.*;
import java.io.*;
import java.time.format.DateTimeFormatter;
import java.time.LocalDateTime;
import java.nio.file.Files;
import java.nio.file.*;

public class LoggingSystem {
    private String logNameFile;
    private String lastExtractionDate;
    private File file;

    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    LocalDateTime now = LocalDateTime.now();

    public LoggingSystem(String fileDir){
        logNameFile = fileDir;
        file = new File(logNameFile);
        now = LocalDateTime.now();
        lastExtractionDate = dtf.format(now);
    }

    //writes to log file
    public void log(String user, String eventData){
        try {
            FileWriter myWriter = new FileWriter(file, true);
            BufferedWriter bw = new BufferedWriter(myWriter);
            now = LocalDateTime.now();
            bw.write(dtf.format(now) + " {" + user + "} " + eventData);
            bw.newLine();
            bw.close();
        } catch (IOException e) {
            System.out.println("Log file " + logNameFile + " not found for logging.");
            e.printStackTrace();
        }
    }
```

```
//deletes log file
public void wipeLogs(){
    try {
        FileWriter myWriter = new FileWriter(file);
        myWriter.write("");
        myWriter.close();
    } catch (IOException e) {
        System.out.println("Log file " + logNameFile + " could not be wiped.");
        e.printStackTrace();
    }
}

//copies log file to location
public boolean extractLogs(String location) {
    try {
        Path copied = Paths.get(location);
        Path originalPath = Paths.get(logNameFile);
        Files.copy(originalPath, copied, StandardCopyOption.REPLACE_EXISTING);
        now = LocalDateTime.now();
        lastExtractionDate = dtf.format(now);
        return true;
    } catch (IOException E){
        System.out.println("Directory not found");
        return false;
    }
}

//displays contents of log
public void viewLogs(){
    try {
        File myObj = new File(logNameFile);
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            System.out.println(data);
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("Log file " + logNameFile + " not found for viewing.");
        e.printStackTrace();
    }
}
```

```
    public String getExtractionDate(){
        return lastExtractionDate;
    }
}
```

RouterSim.java

```
import java.util.*;
import java.io.*;

public class RouterSim {
    private String scriptName;
    private int refreshRate; //in milliseconds
    private int numSensors;
    private File file;
    private int blockCounter;

    ArrayList<String> scriptLines = new ArrayList<String>();

    public RouterSim(String scriptName, int refreshRate, int numSensors){
        this.scriptName = scriptName;
        this.refreshRate = refreshRate;
        this.numSensors = numSensors;
        file = new File(scriptName);
        this.blockCounter = 0;

        this.loadArrayList();
    }

    // returns an array of all sensor data strings that are sent in a period of time
    public String[] getDataBlock(){
        String[] block = new String[numSensors];
        for (int i = 0; i<numSensors; i++){
            try {
                block[i] = scriptLines.get(blockCounter*numSensors+i);
            } catch (IndexOutOfBoundsException E) {
                return null;
            }
        }
        blockCounter++;
        return block;
    }

    //load script into local ArrayList
    private void loadArrayList(){
```

```
        try {
            Scanner myReader = new Scanner(file);
            while (myReader.hasNextLine()) {
                String data = myReader.nextLine();
                scriptLines.add(data);
            }
        } catch (IOException e) {
            System.out.println("Router script not found");
            e.printStackTrace();
        }
    }

    public int getRefreshRate(){
        return refreshRate;
    }
}
```

GPSSensor.java

```
public class GPSSensor extends Sensor {
    private float currentLatitude;
    private float currentLongitude;
    private float cachedLatitude;
    private float cachedLongitude;
    private double duration; //time since last signal in hours

    private double earthRadius = 3958.8; // in miles

    public GPSSensor (String sensorID, String location){
        this.sensorID = sensorID;
        this.location = location;
        this.duration = 1;
        this.currentLatitude = 45;
        this.currentLongitude = 45;
    }

    // listens for input from data which must include coordinate data
    public boolean getSensorData(String sensorData) {
        try {
            cachedLatitude = currentLatitude;
            cachedLongitude = currentLongitude;
            currentLatitude = Float.parseFloat(sensorData.split(" ")[1]);
            currentLongitude = Float.parseFloat(sensorData.split(" ")[2]);
            duration = Double.parseDouble(sensorData.split(" ")[3]);
            return true;
```

```
        } catch (Exception E){
            return false;
        }
    }

    //calculates estimated difference in coordinates
    public double getDistanceDifference(){
        double lat1 = currentLatitude * Math.PI/180;
        double lat2 = cachedLatitude * Math.PI/180;
        double latitudeDifference = (currentLatitude - cachedLatitude) * Math.PI/180;
        double longitudeDifference = (currentLongitude - cachedLongitude) * Math.PI/180.0;
        double a = Math.sin(latitudeDifference/2.0) * Math.sin(latitudeDifference/2) +
                Math.cos(lat1) * Math.cos(lat2) *
                Math.sin(longitudeDifference/2.0) * Math.sin(longitudeDifference/2.0);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
        return earthRadius * c;
    }

    //calculates speed of train based on difference in location
    public double getSpeed(){
        return this.getDistanceDifference() / duration;
    }
}
```

DopplerRadarSensor.java

```
public class DopplerRadarSensor extends Sensor {
    int position; // 1 if positioned the front, 2 if positioned at the back
    double sentFrequency; // frequency of sent signal
    double returnFrequency;
    double duration;
    double speedOfWave = 186282.397; // miles per second

    public DopplerRadarSensor (String sensorID, String location, int position, double frequency){
        this.sensorID = sensorID;
        this.location = location;
        this.position = position;
        this.sentFrequency = frequency;
        this.duration = Double.POSITIVE_INFINITY;
    }

    //listens for input from sensor which must include return frequency and pulse duration
    public boolean getSensorData(String sensorData){
        try {
            returnFrequency = Double.parseDouble(sensorData.split(" ")[1]);
```

```
            duration = Double.parseDouble(sensorData.split(" ")[2]);
            return true;
        } catch (Exception E) {
            return false;
        }
    }

    //calculates distance to object based on duration of pulse
    public double calculateDistance(){
        return duration*speedOfWave;
    }

    //calculate relative speed of object based on
    public double calculateSpeedDifference(){
        return (speedOfWave * (sentFrequency - returnFrequency) / sentFrequency);
    }

    public int getPosition(){
        return position;
    }

    public double getSentFrequency(){
        return sentFrequency;
    }

    public double getReturnFrequency(){
        return returnFrequency;
    }

    public double getDuration(){
        return duration;
    }
}
```

IceFormationSensor.java

```
public class IceFormationSensor extends Sensor {
    private double currentTemperature; // in celsius
    private double currentHumidity; // percentage

    public IceFormationSensor (String sensorID, String location) {
        this.sensorID = sensorID;
        this.location = location;
    }
```

```
    //listens for input from sensor which must include outside temperature and humidity
    public boolean getSensorData(String sensorData){
        try {
            currentTemperature = Double.parseDouble(sensorData.split(" ")[1]);
            currentHumidity = Double.parseDouble(sensorData.split(" ")[2]);
            return true;
        } catch (Exception E){
            return false;
        }
    }

    public double getTemperature(){
        return currentTemperature;
    }

    public double getHumidity(){
        return currentHumidity;
    }

    //checks current conditions if ice formation is possible
    public boolean checkFormation(){
        if ( currentTemperature <= 0.0 && currentHumidity >= 72.0 && currentHumidity <= 78.0) {
            return true;
        }
        return false;
    }
}
```

CrossingGateSensor.java

```
public class CrossingGateSensor extends Sensor {
    private boolean lightDetected;
    private boolean soundDetected;
    private double distanceTo; // in miles

    public CrossingGateSensor(String sensorID, String location){
        this.sensorID = sensorID;
        this.location = location;
        lightDetected = false;
        soundDetected = false;
    }

    //listens for input from sensor which must include RPM
    public boolean getSensorData(String sensorData){
        try {
```

```
            if (Integer.parseInt(sensorData.split(" ")[1]) == 1) {
                lightDetected = true;
            } else {
                lightDetected = false;
            }

            if (Integer.parseInt(sensorData.split(" ")[2]) == 1) {
                soundDetected = true;
            } else {
                soundDetected = false;
            }

            distanceTo = Double.parseDouble(sensorData.split(" ")[3]);

            return true;
        } catch (Exception E) {
            return false;
        }
    }

    public boolean getLightDetected(){
        return lightDetected;
    }

    public boolean getSoundDetected(){
        return soundDetected;
    }

    public double getDistanceTo(){
        return distanceTo;
    }
}
```

WheelSensor.java

```
import java.lang.Math;

public class WheelSensor extends Sensor {
    String wheelID;
    private float wheelDiameter; // in INCHES
    private float currentRPM;

    //constructor
    public WheelSensor(String sensorID, String location, float wheelDiameter){
        this.sensorID = sensorID;
```

```
        this.location = location;
        this.wheelDiameter = wheelDiameter;
    }

    //listens for input from sensor which must include RPM
    public boolean getSensorData(String sensorData){
        try {
            currentRPM = Float.parseFloat(sensorData.split(" ")[1]);
            return true;
        } catch (Exception E){
            return false;
        }
    }

    public float getCurrentRPM(){
        return currentRPM;
    }

    //get current speed based on current rpm
    public double getSpeedFromRPM(){
        return wheelDiameter * Math.PI * currentRPM * 60.0 / 63360.0;
    }
}
```

Sensor.java

```
public class Sensor {
        String sensorID;
        String location;

    public void getSensorData(){}
}
```

7. Test Cases

7.1. Scenario Tests

4.1.1	System logs authorized users in and denies unauthorized ones	PASSED
4.1.2	System logs out a users and requires a username and password for it be active again	PASSED
4.1.3	Technician is able to interact with log files through technician exclusive menu options	PASSED

4.1.5	System analyzes Doppler Radar Sensor data that detects an object and reacts accordingly	PASSED
4.1.6	System calculates potential slippage and reacts accordingly	PASSED
4.1.7	System analyzes Crossing Gate Sensor data that detects crossing gate lights and reacts accordingly	PASSED
4.1.8	System analyzes Crossing Gate Sensor data that detects crossing gate sounds and reacts accordingly	PASSED
4.1.9	System analyzes distance to crossing gate from Crossing Gate Sensor and displays blow horn messages for an appropriate amount of time	PASSED
4.1.10	System analyzes Ice Formation Sensor data that detects optimal ice formation conditions and reacts accordingly	PASSED
4.1.11	Logging System records all events going on in the system	PASSED
4.1.12	System generates inform signal when an inform event occur	PASSED
4.1.13	System generates alert signal when an alert event occurs	PASSED

7.2. Validation Tests

7.1	Automatically connect to sensors on conductor log in	PASSED
7.2	Read data from sensor only when speed is detected to be nonzero from any source	PASSED
7.3	When a pulse of similar frequency is detected, record time it took for response and observed frequency	PASSED
7.4	Send no abnormal signals if calculated distance to detected object is greater than 10 miles	PASSED
7.5	Raise inform message if distance is less than or equal to 10 miles but greater than 5 mile	PASSED
7.6	Raise alert message if distance is less than or equal to 5 mile	PASSED
7.7	Looks up GPS coordinates at set refresh rate	PASSED
7.8	Calculates distance traveled between current and cached coordinates	PASSED
7.9	Calculates speed by dividing distance by refresh rate interval	PASSED

7.10	Calculates detected object's speed by adding doppler speed difference to actual speed	PASSED

7.11	Send alert message when Crossing Gate Sensor detects both gate lights and sounds	PASSED
7.12	Send message for a set amount of seconds to blow horn based on distance to detected crossing gate	PASSED
7.13	At set refresh rate, look up the data from microphone and camera sensors	PASSED
7.14	If gate sensor is detecting a gate, set variable to true	PASSED
7.15	If gate sensor is not detecting a gate, set variable to false	PASSED
7.16	At set refresh rate poll all wheel sensors for rpm, remove outliers and use the average of the remaining values	PASSED
7.17	Calculate speed using wheel diameter and sensor rpm	PASSED
7.18	Send inform signal when speed based on RPM and GPS is greater than 2 mph but less than 5 mph	PASSED
7.19	Send alert signal when speed based on RPM and GPS is greater than 5 mph	PASSED
7.20	At set refresh rate, poll ice sensor for humidity and temperature	PASSED
7.21	Send inform signal when humidity is between 72% and 78% and temperature is below 0C	PASSED

7.22	All inform signals are displayed in yellow text, include signal message, and suggestions on further actions	PASSED
7.23	Inform signals include: minor wheel slippage, objects behind or in front being between 5 and 10 miles of the train, the possibility of ice formation, crossing gate being detected	PASSED
7.24	Wheel Slippage inform messages include the amount of wheel slippage and suggestion to slow down	PASSED

7.25	Object proximity inform messages include the direction in which the object is detected, the distance to it, its speed, and a suggestion	PASSED
7.26	Ice formation message displays current temperature and humidity, and a suggestion to slow down	PASSED
7.27	All alert signals are displayed in red text, include signal message, and suggestion on further actions	PASSED
7.28	Wheel Slippage alert messages include the amount of wheel slippage, and a suggestion	PASSED